



## **Conceptualização e desenvolvimento de uma framework de clustering**

**RICARDO FILIPE FERNANDES TABOADA**

Junho de 2017

# **CONCEPTUALIZATION AND DEVELOPMENT OF A CLUSTERING FRAMEWORK**

**Ricardo Filipe Fernandes Taboada**

**Dissertation to obtain the Master's Degree in Informatics Engineering,  
specialization in Computer Systems**

**Advisor: Dr. Paulo Gandra de Sousa**

**Jury:**

Chairman:

Vowels:

Porto, Junho 2017



# Dedictory

This thesis is dedicated to my fiancée, Paula, who always supported and motivated me along this journey.



# Resumo

Com a proliferação de todo o tipo de serviços baseados em plataformas digitais, como por exemplo, o e-commerce o home banking ou mesmo as redes sociais, o conceito de sistemas distribuídos ganhou um novo folgo, e com ele, surgiram novas necessidades de se atingir altos níveis de disponibilidade para determinados sistemas de software. Este cenário obriga a que as infraestruturas tecnológicas atuais incluam várias réplicas desses mesmos sistemas, de forma a manter o serviço sempre disponível ainda que ocorra uma falha num ou noutro sistema. A maior parte dos sistemas atuais incluem duas camadas distintas, a camada aplicacional, onde corre a lógica de negócio, e a camada de persistência onde os dados são guardados de forma não volátil. Embora, normalmente, de forma simples se consigam replicar os aplicativos desses sistemas, replicar as camadas de persistência revela-se a maior parte das vezes um desafio bem mais complexo.

Esta dissertação apresenta um problema concreto de uma necessidade de aplicar replicação de dados num sistema distribuído que se encontra atualmente em ambiente de produção, de forma a poder garantir-se a disponibilidade do mesmo. Do estudo realizado sobre os principais conceitos de replicação de dados, assim como algumas frameworks de replicação a nível de middleware, e o problema em questão, foi possível conceptualizar e desenvolver uma nova framework de clustering ao nível do middleware que pode ser aplicada em sistemas aos quais se queira adicionar capacidade de clustering, independentemente do tipo de persistência com os quais os mesmos interagem.

**Palavras-chave:** Clustering, Replicação otimística de dados, Alta disponibilidade, Consistência Eventual, Código aberto



# Abstract

With the proliferation of all kinds of services based on digital platforms, as for example, the e-commerce, the home banking or even the social networks, the concept of distributed systems gained a new breadth, and with it, appeared new necessities to achieve higher levels of high availability in some specific software systems. This scenario forces the need of the actual technological infrastructures to include several replicas of those systems, in order to ensure the service availability, even in an advent of a failure in one or more systems. The majority of the actual systems include two distinct layers, the application layer, where the business logic runs, and the persistence layer, where the data is stored in a non-volatile way. Although, usually, is simple to apply replication to the application layer of those systems, applying replication on the persistence layers reveals itself most of the times a much more complex challenge.

This master thesis presents a concrete problem of the necessity to apply data replication to a distributed system that is currently in a production environment, in order to ensure its availability. Through study performed both on the main concepts of data replication, as on some middleware based replication frameworks, and taking into the account the problem in hand, it was possible to conceptualize and develop a new middleware clustering framework that can be applied to systems to which is wanted to add clustering capabilities, regardless of the persistence type they interact with.

**Keywords:** Clustering, Optimistic data replication, High Availability, Eventual consistency, Open source





# Acknowledgements

I would like to thank my employer, Porto Tech Center for the opportunity to develop this work in such a challenging subject.

I would also like to thank my advisors, Dr. Paulo Gandra de Sousa and Eng. Hugo Conceição for the help and support through the development of this work.



# Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Background .....	1
1.2	Thesis Subject and Motivation .....	2
1.3	Main goals.....	2
1.4	Scope .....	3
1.5	Success Criteria.....	3
1.6	Evaluation .....	4
1.6.1	What will be evaluated.....	4
1.6.2	Which metrics will be used .....	5
1.6.3	Hypothesis .....	5
1.7	Value Analysis .....	6
1.8	Expected Contributions .....	12
1.9	Document Structure .....	12
<b>2</b>	<b>Context.....</b>	<b>15</b>
2.1	Current Scenario.....	15
2.1.1	Functional Requirements .....	20
2.1.2	Non Functional Requirements.....	20
<b>3</b>	<b>State of the Art .....</b>	<b>23</b>
3.1	Background Concepts .....	23
3.1.1	High Availability .....	23
3.1.2	Database Consistency.....	24
3.1.3	Database Replication .....	28
3.2	Existing Frameworks .....	31
3.2.1	C-JDBC .....	31
3.2.2	Ganymed.....	34
3.2.3	Tashkent .....	36
3.2.4	Neo4j High Availability .....	38
3.2.5	Conclusion .....	39
<b>4</b>	<b>Proposed Solution .....</b>	<b>41</b>
4.1	Introducing Replic8 .....	41
4.2	Replic8 High Level Transaction Replication Flow.....	41
4.3	High level Architecture Overview .....	42
4.3.1	Transaction Module .....	42
4.3.2	Recovery Logger Module .....	44
4.3.3	Cluster Module.....	44
4.3.4	Health Module .....	45

4.4	Design Approaches and Decisions .....	46
<b>5</b>	<b>Development .....</b>	<b>49</b>
5.1	Design Patterns .....	49
5.1.1	Creational Design Patterns .....	49
5.1.2	Structural Design Patterns .....	49
5.1.3	Behavioral Design Patterns .....	50
5.2	Unit and Integration Testing .....	50
5.3	Replic8 Tech Stack .....	50
5.3.1	Why Erlang for communication? .....	51
5.4	Replic8 Modules Breakdown .....	52
5.4.1	The Transaction Interceptor .....	53
5.4.2	Transaction Broadcaster .....	56
5.4.3	Remote Transaction Processor .....	59
5.4.4	Persistence Version Service .....	61
5.4.5	Transaction Recovery Logger .....	61
5.4.6	Cluster Registry .....	62
5.4.7	Cluster Context .....	65
5.4.8	Master Failover .....	65
5.4.9	Health Check .....	66
5.4.10	Persistence Version Convergence .....	66
5.4.11	Senders and Receivers .....	67
5.5	Replic8 Properties .....	69
5.5.1	Recovery Log properties .....	69
5.5.2	Cluster properties .....	69
5.6	Limitations .....	70
<b>6</b>	<b>Validation .....</b>	<b>73</b>
6.1	Validation infrastructure .....	73
6.1.1	Network Topology .....	73
6.1.2	Hardware specifications .....	73
6.2	Validation scenarios .....	74
6.2.1	A1: Write throughput Replic8 impact .....	74
6.2.2	A2: Write throughput slave instance increase impact .....	75
6.2.3	B1: Evaluation of the final state of all the database instances .....	78
6.2.4	B2: Slave persistence convergence time .....	79
6.2.5	C1: Cluster behavior when a slave instance goes offline .....	83
6.2.6	C2: Cluster behavior when the master instance goes offline .....	84
6.3	Conclusion .....	85
<b>7</b>	<b>Conclusions .....</b>	<b>87</b>
7.1	Open Issues and Future Work .....	88

# List of Figures

Figure 1 - System PC Technological Stack .....	16
Figure 2 – Logistics Support System High Level Architecture .....	17
Figure 3 - Create Node Sequence Diagram .....	18
Figure 4 - Transaction Deadlock UML sequence diagram (Liu & Zsu, 2009).....	25
Figure 5 - C-JDBC component architecture (Cecchet et al., 2004).....	33
Figure 6 - Ganymed component architecture (Plattner & Alonso, 2004).....	35
Figure 7 - Tashkent-MW component architecture (Elnikety et al., 2006) .....	37
Figure 8 - Tashkent transaction certification flow (Elnikety et al., 2006) .....	38
Figure 9 - Neo4J-HA component architecture (Montag, 2013) .....	39
Figure 10 - High level transaction replication flow UML activity diagram .....	41
Figure 11 - Replic8 High level UML component diagram .....	42
Figure 12 - Transaction Module UML component diagram .....	43
Figure 13 - Recovery Logger module UML component diagram .....	44
Figure 14 - Cluster Module UML component diagram .....	45
Figure 15 - Health Module UML component diagram .....	46
Figure 16 - Replic8 as a Neo4J wrapper UML component diagram.....	46
Figure 17 - Replic8 typical positioning in the application UML component diagram .....	47
Figure 18 - Replic8 Technology stack .....	51
Figure 19 - Replic8 high level component overview .....	52
Figure 20 - Transaction Interception with Replic8 .....	53
Figure 21 - AspectJ weaving overview .....	54
Figure 22 - Transaction Interceptor UML class diagram .....	54
Figure 23 - Transaction Interceptor flow UML activity diagram .....	55
Figure 24 - Transaction Interception involved classes UML class diagram.....	56
Figure 25 - Observer Registration Flow UML activity diagram .....	57
Figure 26 - Transaction Broadcaster UML class diagram .....	58
Figure 27 - Transaction Broadcast flow UML activity diagram .....	59
Figure 28 - Remote Transaction Processor flow UML activity diagram .....	60
Figure 29 - Remote Transaction Processor UML class diagram .....	60
Figure 30 - Persistence version convergence flow UML activity diagram .....	61
Figure 31 - Transaction Recovery Logger UML class diagram.....	62
Figure 32 - Replic8 initialization as Master UML sequence diagram .....	64
Figure 33 - Cluster context configuration file example.....	65
Figure 34 - Persistence version convergence UML sequence diagram.....	67
Figure 35 - Replic8 senders UML class diagram .....	68
Figure 36 - Replic8 receivers UML class diagram .....	68
Figure 37 - Replic8 properties file .....	69
Figure 38 - Replic8 validation infrastructure setup.....	73
Figure 39 - Slave disconnection from the cluster.....	84
Figure 40 - Master Failover to next in hierarchy slave.....	85

Figure 41 - Canvas Model ..... 93

# List of Tables

Table 1 - Availability Categories (Gray & Siewiorek, 1991).....	23
Table 2 - Infrastructure hardware specifications.....	73
Table 3 - Parametric T test between measured mean throughput slowdown and $H_0$ value ...	75
Table 4 - T test between the measured mean throughput slowdown with two slaves and $H_0$ value.....	77
Table 5- T test between the measured mean transaction slowdown with three slaves and $H_0$ value.....	78
Table 6 - Slave One convergence times against $H_0$ limit value for a two slave cluster .....	80
Table 7 - Slave Two convergence times against $H_0$ limit value for a two slave cluster .....	80
Table 8 - Slave One convergence times against $H_0$ limit value for a three slave cluster .....	82
Table 9 - Slave Two convergence times against $H_0$ limit value e for a three slave cluster .....	82
Table 10 - Slave Three convergence times against $H_0$ limit value e for a three slave cluster ..	83
Table 11 - A/B test throughput for SystemPC w/o and with Replic8.....	94
Table 12 - A/B test throughput for Replic8 with one and two slaves .....	95
Table 13 - A/B test throughput for Replic8 with two and three slaves .....	96
Table 14 - Persistence convergence verification results.....	97
Table 15 - Persistence convergence times for a two slave cluster .....	98
Table 16 - Persistence convergence times for a three slave cluster.....	99





# Acronyms and Nomenclature

## Acronyms list

<b>1SR</b>	<i>One-Copy Serializability</i>
<b>2PC</b>	<i>Two-Phase Commit</i>
<b>ACID</b>	<i>Atomicity, Consistency, Isolation and Durability</i>
<b>AJC</b>	<i>AspectJ Compiler</i>
<b>AOP</b>	<i>Aspect Oriented Programming</i>
<b>API</b>	<i>Application Interface</i>
<b>AWS</b>	<i>Amazon Web Services</i>
<b>ARDS</b>	<i>Amazon Relational Database Service</i>
<b>BASE</b>	<i>Basically Available, Soft state Eventual Consistency</i>
<b>C-JDBC</b>	<i>Clustered JDBC</i>
<b>CAP</b>	<i>Consistency, Availability and Partition tolerance</i>
<b>CCC</b>	<i>Cluster Context Component</i>
<b>CMDVC</b>	<i>Conceptual Model for Decomposing the Value for the Customer</i>
<b>CPV</b>	<i>Current Persistence Version</i>
<b>CR</b>	<i>Cluster Registry</i>
<b>CUD</b>	<i>Create Update and Delete</i>
<b>DDBS</b>	<i>Distributed Database System</i>
<b>DSS</b>	<i>Decision Support System</i>
<b>HA</b>	<i>High Availability</i>
<b>HTTP</b>	<i>Hypertext Transfer Protocol</i>
<b>IDE</b>	<i>Integrated Development Environment</i>
<b>ISEP</b>	<i>Instituto Superior de Engenharia do Porto</i>
<b>JDBC</b>	<i>Java DataBase Connectivity</i>
<b>JDK</b>	<i>Java Development Kit</i>

<b>JMS</b>	<i>Java Message Service</i>
<b>JMX</b>	<i>Java Management Extension</i>
<b>JVM</b>	<i>Java Virtual Machine</i>
<b>LAN</b>	<i>Local Area Network</i>
<b>LN</b>	<i>Logistics Network</i>
<b>MFC</b>	<i>Master Failover Component</i>
<b>MVC</b>	<i>Model View Controller</i>
<b>MVCC</b>	<i>Multi Version Concurrency Control</i>
<b>NoSQL</b>	<i>Non Structured Query Language</i>
<b>OLTP</b>	<i>On-Line Transaction Processing</i>
<b>OOP</b>	<i>Object Oriented Programming</i>
<b>POJO</b>	<i>Plain Old Java Object</i>
<b>PV</b>	<i>Persistence Version</i>
<b>PVS</b>	<i>Persistence Version Service</i>
<b>PVCC</b>	<i>Persistence Version Convergence Component</i>
<b>RAID</b>	<i>Redundant Array of Inexpensive Disks</i>
<b>RAIDb</b>	<i>Redundant Array of Inexpensive Databases</i>
<b>RDBMS</b>	<i>Relational Database Management System</i>
<b>REST</b>	<i>Representational State Transfer</i>
<b>RMI</b>	<i>Remote Method Invocation</i>
<b>RSI-PC</b>	<i>Replicated Snapshot Isolation with Primary Copy</i>
<b>RTP</b>	<i>Remote Transaction Processor</i>
<b>SI</b>	<i>Snapshot Isolation</i>
<b>SPOF</b>	<i>Single Point Of Failure</i>
<b>SOAP</b>	<i>Simple Object Access Protocol</i>
<b>SQL</b>	<i>Structured Query Language</i>

<b>TI</b>	<i>Transaction Interceptor</i>
<b>TB</b>	<i>Transaction Broadcaster</i>
<b>TPC</b>	<i>Transaction Processing Performance Council</i>
<b>TRL</b>	<i>Transaction Recovery Logger</i>
<b>UML</b>	<i>Unified Modeling Language</i>
<b>VC</b>	<i>Value for the Customer</i>
<b>VNA</b>	<i>Value Network Analysis</i>
<b>WAN</b>	<i>Wide Area Network</i>

## **Symbols list**

<b><math>\alpha</math></b>	alpha
<b><math>\Delta</math></b>	delta



# 1 Introduction

## 1.1 Background

Nowadays due to the appearance of new technologies and the need for more software modularity, Distributed Systems aim to leave the traditional monolithic approach. Instead, processing is delegated to several application services that compose the whole system (Coulouris et al., 2005; Tanenbaum & Van Steen, 2002).

As the number of requests to each of these services increases, it grows the need to scale them either vertically or horizontally. This need is derived from the necessity of achieve not only increased performance but also having redundant fault tolerable systems that are always available to the end users, even in the advent of a failure on one of the system component, and therefore providing High Availability (HA) (Fox & Brewer, 1999).

With vertical scaling or Scale-up (Brebner & Gosper, 2003; Michael et al., 2007) the machine resources where one of the system services is running, can be continually increased until the desired performance is achieved. Although vertical scaling can be a short term solution, the system will become bottlenecked for example in terms of networking capacity, and induce very high costs to obtain a little performance increase. Also with vertical scaling, although application redundancy can be achieved having several application instances running in one machine, if the machine fails, all instances will go down, which depending on the business type and/or requirements, may be simply not acceptable.

Horizontal scaling or Scale-out (Brebner & Gosper, 2003; Michael et al., 2007) on the other hand provides a good level of redundancy and decreased costs to scale compared to vertical scaling. In horizontal scaling one or more services are replicated, through several instances of that service. Each instance of the service runs on its own machine. Theoretically as the need for more processing power increases, more instances can be added to the system cluster, and therefore there's not really a limitation for the number of instances that can be added.

For many applications, horizontal scaling seems a good fit, but it does not come without its drawbacks. One of the bigger problems with horizontal scaling is data persistence.

When the service does not persist any kind of data, it just processes some data and gives a response back. The requesting client does not know which instance processed the data, and it really does not need to know it, as long as it receives a response. Other instances of the called service also don't need to know that one of them got called.

The biggest problem arises when the replicated services perform data persistence. If there are two or more instances of a service with its own persistence instance, any create, update or delete (CUD) operation on one instance must be replicated to the others. Depending on the business needs, it could be needed to enforce Strong Consistency (Fox & Brewer, 1999), which means that if data updated in one instance fails to update on another instance, the operation has to be atomically reverted. Two-phase commit protocol or other type of synchronous replication, like distributed transactions can be used to enforce an "all or nothing" approach and rollback the whole transaction in case of an update failure in one node, ending up with old data. Sometimes this is simply not acceptable as it does not conform to the business requirements, not to talk that distribution transactions should only be used in very constrained scope as they simply do not scale (Gray et al., 1996; Helland, 2007).

## **1.2 Thesis Subject and Motivation**

This master thesis aims to solve the need to add database clustering to a real world application developed In Porto Tech Center which is backed up by Neo4J (Neo4J, n.d.), a non-relational graph database. Although Neo4J offers an Enterprise Edition, which addresses this need, as of now the cost involved for the upgrade are too high for the short term benefits.

If by one side, replication for relational databases like PostgreSQL (PostgreSQL, n.d.) has been subject for many studies over the years, non-relational database are quite newer, and as such it didn't get as much attention. The work described in this document also tries to address data replication problems for HA Distributed Systems, researching what has been done before in this field and offering an alternative middleware clustering solution that is not only applicable to Neo4J backed applications, but also to other types of both non-relational and relational databases.

## **1.3 Main goals**

The work developed in this master thesis has the objective of implementing a framework to enable clustering characteristics to a service or application with data persistence.

One of the main goals to achieve is data consistency across replicated systems, while trying to keep them as up to date as possible in terms of persisted data.

The goals of this work are:

- Perform a theoretical study on data replication;
- Identify the main data replication techniques in use on both open source and commercial systems;

- Research on what has been already done in the data replication field, and its relation with the problem presented;
- Define and draft specifications for a new data replication library/component, whose responsibility is to articulate persistence access for a number 'n' of database instances.
- Develop and implement a prototype library/component which complies with the defined specifications;
- Perform a technical validation of the prototype.

## 1.4 Scope

The scope of this work is to build a viable data replication solution and therefore this document focuses on this area. Upon development of the solution, a simulated scenario with a non-relational database will be setup.

Related issues like, for example, load balancing will be addressed, but not included in this document.

## 1.5 Success Criteria

The success criteria is directly linked to the objectives defined in 1.3, as such for each objective, a success criteria is defined:

- Perform a theoretical study on data replication:
 

Before starting developing and implementing a solution, it is important to acquire essential knowledge about data replication, its advantages, and pitfalls. Therefore the success criteria will be measured through the number, relevance, and age of the bibliographic references used.
- Identify the main data replication techniques in use on both open source and commercial systems:
 

Identification of the several data replication techniques is essential to identify which one will be the most adequate for the exposed problem. Therefore, up-to-date information on existing techniques and its characteristics should be conveniently documented.
- Research on what has been already done in the data replication field, and its relation with the problem presented:
 

Several studies and solutions had already been proposed over the years. A study should be carefully taken on data replication middleware services, either proposed and/or in production and its applicability to the problem exposed.



- Define and draft specifications for a new data replication library/component, that can address both relation and non-relational databases:

Before building a new data replication system, a series of specifications should be carefully defined. The quality and the detail of the specifications will ensure the correctness of the developed software. Therefore, the specifications presented should be detailed and concise.

- Develop and implement a prototype library/component which complies with the defined specifications:

After specifications are written, a prototype must be developed. The success of the prototype will be dependent on the fulfillment of each specification, and its applicability to the problem.

- Perform a technical validation of the prototype:

Technical validations should be performed to ensure both data consistency and performance across replication systems. More details on the specific validations to be performed can be read in chapter 1.6 Evaluation.

## 1.6 Evaluation

The proposed solution will be evaluated according to three main axis, performance, consistency and availability.

Porto Tech Center's System PC will be the targeted system to which the proposed solution will be applied. As stated earlier this system uses a Neo4J, a non-relational database to persist data.

The main entities of System PC are Nodes and Relationships. Those are the two types of entities that will be persisted in System PC database and therefore will be targeted for clustering. For confidentiality reasons, the structure of those two entities will not be exposed.

### 1.6.1 What will be evaluated

- **A1:** Write Throughput: The rate at which write operations are performed on the system. It is important to verify the system behavior regarding write operations when replication coordination is being applied. This will use an A/B type test, where the subject application SystemPC will be tested both before implementing the Replic8 framework and after in order to evaluate the performance impact.
- **A2:** Write Throughput: Evaluate the degradation in write throughput between a cluster with one, two or three slave instances. Testing clusters with different sizes is important to identify possible bottlenecks.
- **B1:** Data Consistency: Evaluation of the final state of all the database instances. At the final of the closed loop, all database instances should be consistent between each

other. The time it takes for consistency convergence can vary from slave to slave. Nevertheless they all should eventually converge to the same state.

- **B2:** Data Consistency: How long does it takes to a slave instance to be consistent with the master instance?
- **C1:** Service Availability: Randomly shut down one or more slave instances and verify that the system continues to successfully replicate transactions to the remaining instances.
- **C2:** Service Availability: Shut down the master instance and verify that the failover is successfully handled by the cluster. Another slave should assume the master role and as such the cluster should continue to accept write requests.

### 1.6.2 Which metrics will be used

- **A1; A2:** Write throughput will be evaluated using continuous data in the form of requests per second (**req/s**).
- **B1:** Data Consistency will be evaluated with binomial data (true/false). At the end of each test sample, either there is data consistency or not.
- **B2:** Data Consistency convergence acceptable time will be evaluated using both continuous data measuring the persistence convergence time and binomial data (true/false). At the end of each test sample, for each slave, either data convergence was performed within the predefined accepted time threshold or not.
- **C1; C2:** Service Availability will be evaluated with binomial data (true/false). During each test, findings will be recorded asserting if the behavior is correct.

### 1.6.3 Hypothesis

Taking evaluation points from chapter 1.6.1, a series of hypotheses are formulated to assert the correct functioning of the framework. First, a hypothesis called the 'null' hypothesis ( $H_0$ ) is formulated asserting the unwanted behavior, then an alternative hypothesis ( $H_a$ ) is developed stating the opposite, asserting the expected behavior.

#### 1.6.3.1 A1

- $H_0$  – The write throughput of the application is slowed down more than 25% when Replic8 is configured to handle transaction replication.
- $H_a$  – The write throughput of the application is not slowed down by more than 25% by the Replic8 clustering framework.

#### 1.6.3.2 A2

- $H_0$  – The write throughput is slowed down by more than 10% when an instance is added to the cluster.
- $H_a$  – The write throughput should not be slowed down by more than 10% for each instance added to the cluster.

#### 1.6.3.3 B1

- $H_0$  – Data inconsistency is spotted in one or more instances after the test run.
- $H_a$  – After a test run, all instances will be consistent with each other regarding the persisted data.

#### 1.6.3.4 B2

- $H_0$  – Data convergence for at least one instance is not achieved within a time frame representing 25% of total test run time.
- $H_a$  – After a test run, all instances should converge to a consistent state with the master instance within a time frame representing 25% of total test run time.

#### 1.6.3.5 C1

- $H_0$  – After shutting down one or more slave instances, the cluster behavior is affected or not able to perform transaction replication at all.
- $H_a$  – When one or more slave instances are shutdown, the cluster continues behave the same way and handling transaction replication successfully.

#### 1.6.3.6 C2

- $H_0$  – When the master instance is shut down, the failover is not successful and the cluster no longer accepts writes.
- $H_a$  – After shutting down the master instance, the failover is successfully performed and another instance immediately assumes the master role, ensuring the cluster remain available and accepting writes.

## 1.7 Value Analysis

The value analysis serves the purpose of overviewing the benefits of a product, being it tangible or not. More specifically “The primary objective of value analysis is assess how to

increase the value of an item or service at the lowest cost without sacrificing quality” (Nicola, 2015).

The value proposition of this master thesis is to enable High Availability capabilities for distributed systems with data persistence. Therefore the objective is to achieve a uniquely broader spectrum, offering a clustering framework that not only works for applications backed by non- relational databases as Neo4J database but also with other types of databases, being either relational or not. As such the deliverables in this master thesis should represent added value to anyone who wants to enable add High Availability to their software backend applications.

This project appears from a necessity from Porto Tech Center for a specific technical solution. One can easily relate how the opportunity to develop this project appeared with the New Concept for Development Model (NCD), which is used to expose the key components of the Front End innovation, by providing a common language and their definition (Koen et al., 2001).

The opportunity identification appeared from the technical need of a clustering solution which was both technically and financially viable. An opportunity analysis was performed, stating that, although there are already several solutions to address clustering in the market, they neither fit the technology nor budget constraints of Porto Tech Center for its specific needs. According to (Koen et al., 2002), methods usually used both in opportunity identification and analysis are “[...] roadmapping, technology trend analysis and forecast, competitive intelligence analysis, customer trend analysis, market research and scenario planning”. Roadmapping, as such as technology trend and analysis were the main methods used in this specific case.

The Idea Generation and Enrichment started to be developed when it was stated the opportunity to address the need of Porto Tech Center could be theme for this master thesis. From the methods and techniques enumerated by (Koen et al., 2002), market and business needs, identify new technology solutions and an organizational culture that promotes and allows ideas and concept testing were the most used during this process.

Idea Selection was performed by evaluating the technology stack currently in use by Porto Tech Center for the product that will be targeted by the clustering framework. Regarding the methods and techniques for the idea selection, portfolio methodologies based on multiple factors, formal idea selection providing feedback to all idea submitters and the use of the options theory for projects evaluation, are three of the best known techniques (Koen et al., 2002).

The phase of Concept Definition as perceived form NCD model, in this case, is justified by the perfectly identified internal need of a clustering framework that can be applied to our current technologic stack. Some effective techniques to support concept definition are, the quick evaluation of the innovation potential, involving the costumer in an early stage of product testing, establish partnerships with other entities that could better support processes out of the main areas of the competence of the company amongst others (Koen et al., 2002).

As with every business, if fresh innovative ideas are presented in the form of products or processes, they should be transmitted to both existing and potential customers. Therefore it is essential that the benefits of the offered solutions are clear to customers. With this in mind, it

becomes clear the importance of defining a value proposition that clearly targets the market segment that the product wants to reach.

The value proposition can be seen as “an overall view of a company's bundle of products and services that are of value to the customer.” (Osterwalder, 2004). There are two key aspects in the previous statement about value proposition, value, and perceived value, the last one representing the value for the customer. Value can be defined in different forms as states (Nicola et al., 2012) “Value has been defined in different theoretical contexts as need, desire, interest, standard/criteria, beliefs, attitudes, and preferences”.

Value, as the business defines it, is important, “the creation of value is key to any business, and any business activity is about exchanging some tangible and/or intangible good or service and having its value accepted and rewarded by customers or clients, either inside the enterprise or collaborative network or outside.” (Nicola et al., 2012), nevertheless, the perceived value is also another key point for the success of a product or a service. This is the value as the customer sees it and is what defines the desirability of a product or service to a customer, hence “perceived value is the consumer’s overall assessment of the utility of a product based on perceptions of what is received and what is given” (Zeithaml, 1988).

In the case of this master thesis, its aim is to offer a solution to help software developers to better build highly available scalable systems supported by any database technology as long as they can cope with eventual consistency. The final solution to develop is the result of a negotiation between the proponent entity (Porto Tech Center), Instituto Superior de Engenharia do Porto (ISEP) and the authors of this work. As stated by (Filzmoser & Vetschera, 2008), “Negotiations are dynamic processes in which the parties involved communicate to exchange offers, make concessions, raise threats, or otherwise influence each other in order to reach an agreement”. There are several models / scenarios of negotiation:

- Win-Win: In this scenario all involved parties win, benefiting from the final outcome of the negotiation. The involved parties reach an agreement point that will benefit all actors, and no one is at loss (Carnevale & Pruitt, 1992);
- Win-Lose: This negotiation model implies that after negotiations, the final agreement does not totally satisfy one side;
- Lose-Lose: The outcome of the negotiations in this scenario is that both parties remain unsatisfied after the process;
- Triple-Win: This negotiation scenario is comprised by three parties, the customer, the provider and a neutral management program that acts as a mediator. It is the responsibility of the management program to achieve a win situation for both parties. (Lieberman et al., 1997).

In the specific case of this work, a Win-Win scenario was used, and the outcome satisfied all the involved parties. First the proponent entity (Porto Tech Center) will benefit from the outcome of this project, as the needs for a replication solution for their products are addressed. Secondly, ISEP, also benefits from the value added by adding another scientific work to their internal scientific library. Last but not least, the authors of this work will gain from the enormous knowledge derived from its execution.

In Annex A, is attached the possible canvas model for this project. Analyzing the model the key partners identified represent the software vendors that maintain and develop the programming languages and frameworks used in this work:

- Oracle: Maintainer of the Java Programming language;
- Pivotal: Maintainer of Spring Framework;
- Open source Community: Everything around Java is mainly related with open source in mind.

The key activities involved in the execution of this project:

- Research: A research has been performed to study several approaches and existing solutions. This activity represents the documentation contained in the State of the Art chapter of this document;
- Software Architecture & Design: This activity refers to the process involved in designing the solution to address the exposed problem;
- Software Development: All the activities related to the development of the solution presented in this work;
- Software Testing: Test the developed software is a key activity when making quality software;
- Support: Customer support is one of the activities that justify the paid version versus the free version.

The key resources identified are as follow:

- Software Developers: The people responsible for programming the solution;
- Workstations: The computers used by the software developers;
- Software: The software needed to support the development activity, as operating system, programs to design Unified Modeling Language (UML) diagrams, office programs, Integrated Development Environment (IDE) programs, and so on;
- Installations: Space to be use by developers when writing the software. It must provide the essential commodities as electricity, water, internet connection and so on.

The Value propositions this work tries to address are:

- Increase Availability: Opportunity to increase availability with a solution that can manage replicated services;
- Increase Performance: With more service replicas, read performance is increased as the load is distributed between them;

- Database Agnostic: The proposed solution aims to be implemented at the middleware level, abstracted from the database technology used;
- No change to the existing DB schemas: No need to change current DB schemas.

The customer relationships are provided via the following channels:

- Self Service: The official site for the project will be available to everyone looking for information on the project, submit suggestions and also download the framework/component;
- Software development community: Plays a major role in spread new technologies;
- Improvements requests: Customers/Users can provide opinions and ideas on how to improve the software;
- Support: When there is a problem covered by the paid version contract.

The channels through which the project can be obtained are:

- Project Web Site: Can redirect to GitHub or Maven Central;
- GitHub: Where developers can get the source code;
- Maven Central: Main repository for developers looking for the project dependency;
- Forked Software: Some third party software that used or forked the software developed in this project;

The targeted customer Segments are:

- Software Developers;
- Software Development Companies;

The Structure costs are mainly the follow:

- Research & Development: Costs with development and research time;
- Installations: Costs with the installations;

At an initial phase, before the project reaches a predefined maturity level, it can be freely distributed, nevertheless as the product evolves it can be turned into a commercial product deriving it from the free version with more enterprise oriented features. As such the revenue streams are:

- Free version: Can be seen as the initial product, or a more mature product with strip down features;
- Paid version: Can be the fully featured product, derived from the free product, but with more enterprise oriented features, and with technical support.

Measuring added value for customers is not a straight forward task, nevertheless some techniques can be used. These techniques can rely on quantitative methods and/or conceptual models like Value Network Analysis (VNA) (Allee, 2008) or Decomposing the Value for the Customer (CMDVC) (Nicola et al., 2014).

The VNA gives a high level overview of the company, its key partners in the value chain, its relations and the customer. One of the key features of the model is to represent the tangibles and intangibles to the customer (Nicola et al., 2014).

According to Woodall, "Value for the customer (VC) is any demand-side, personal perception of advantage arising out of a customer's association with an organization's offering, and can occur as reduction in sacrifice; presence of benefit (perceived as either attributes or outcomes); the resultant of any weighed combination of sacrifice and benefit; or an aggregation, over time, of any or all these" (Woodall, 2003). Woodall also defined five forms of VC (Woodall, 2003):

- Net VC: This is the perceived value for the customer as a balance of benefits and sacrifices. This balancing will translate in more or less VC;
- Marketing VC: The customer perception of a product or service attributes;
- Sale VC: VC Perhaps the most easily identifiable VC by the customer. Sale VC is directly related to the product price;
- Rational VC: This VC takes a combination of factors, first the customer establishes what interval of prices it will accept to pay for a product, then, it relates this price interval with his perception of the product value based on the perceived attributes;
- Derived VC: Customer perception of value based on other customers experiences with the same product.

Woodall also identified that VC could be distinguished in four temporal positions (Woodall, 2003):

- Ex Ante VC: Pre-purchase, is related to desired and expected values before the purchase;
- Transaction VC: This phase happens at the point of trade. This represents the perceived value for the customer of the value acquired during the transaction;
- Ex Post VC: Happens after the purchase, and is the perception of the received value by the customer;
- Disposition VC: This is the phase when the user intends to dispose or sale the product.

The CMDVC uses Woodall's forms of value and temporal positions, along with the concept of value network for identification of both tangible and intangible assets. These properties are then combined in a quantitative model using techniques derived from operations research like Multi-Criteria Decision Making (Nicola et al., 2014). The CMDVC is modeled following a sequence of three steps, before reaching the enterprise Value Proposition. In the first step a



VNA is performed to identify both tangible and intangible deliverables, and also endogenous and exogenous assets. Each of these deliverables is then related to Woodall's forms of value.

At the first step is possible to obtain an idea on how internal people perceive the identified assets relevance and also their relations with perceived benefits and sacrifices (Nicola et al., 2014).

The second step uses Woodall's temporal positions to obtain information from the enterprise and customer for a specific time position regarding the perception of benefits and sacrifices. The importance of this step relies on selecting the most relevant assets to use as the base on how the customer perceives the Value Proposition of the enterprise (Nicola et al., 2014).

The last step for the construction of the CMDVC model is to combine both the enterprise and customer perspectives from the previous steps to support the assessment of the Value Proposition.

## **1.8 Expected Contributions**

The scope of this work is to specify and develop a new data replication middleware clustering framework/component, evaluate its performance and efficiency and compare it with other solutions.

The following should be considered the expected contributions within this work:

- Up to date state of the art overview providing useful information and guidelines to help conceptualizing and developing a clustering framework.
- Compiled information about other replication frameworks, their advantages and disadvantages;
- A set of specifications, functional, and non-functional requirements written to guide the development of a reliable middleware clustering framework;
- A prototype software complying with both the requirements and specifications, following the defined architecture, offering a solution that can both work with applications backed by relational and non-relational databases.
- Documentation on how to use the software;
- Documented performance and data consistency with the prototype clustering framework applied to the target system.

## **1.9 Document Structure**

The contents of this master thesis are structured as follows:

The first chapter introduces the problem and the work to be done. It contains the introduction to this work, the objectives, the scope and success criteria's and, the validation scenarios to technically validate the proposed solution and the expected contributions. A value analysis is also described in this chapter.

The contextualization and background information that served as the basis of this master thesis is exposed in the second chapter.

The third chapter covers the state of the art and introduces some previous work done in the field to address the same or similar problems. It will empathize primarily in replication middleware solutions.

In chapter four is presented a high level overview of the proposed solution, both with descriptive information and graphical representations of the overall architecture and its components.

Chapter five describes the development phase of the prototype application developed in the context of this master thesis.

The technical validation of the solution is described in chapter six, taking the evaluation scenarios described in chapter one, and asserting the test results.

Finally, final considerations, conclusions and future work are all addressed in chapter seven.

Bibliography and references used in this document are placed after chapter seven.

Annexes can be found at the end of the document.



## 2 Context

### 2.1 Current Scenario

The scenario that drives the motivation for this master thesis is presented in the following paragraphs. Although the subsystem described in the next pages is part of a larger infrastructure, the surrounding context is not relevant for the description and discussion of the problem.

In the area of logistics there are many interesting problems to solve when developing a decision support system (DSS). Specifically, route planning and optimization is a problem that should be addressed carefully as it is the main activity for a logistic company, and as such, bad planning choices can become very costly.

Around route planning there are two main entities, Warehouses and Connections. Warehouses can have many properties, like cold or inflammable storage, capacity and so on. Connections can also share some of those Warehouse properties but also have specific routing properties like transportation method, travel duration, cost, etc. The group of Warehouses and Connections forms the Logistics Network (LN).

A Route is composed by several Connections and defines the path a package is going to travel between one or more Warehouse before reaching the customer. There are several constraints that can be enforced to get the best path, like cost or maximum arrival time. These constraints are calculated from the aggregation of all Connection level constraints for each path combination.

To help route planning, two systems have been built. The first, identified by System LNC (Logistics Network Creator), is responsible to provide the network manager of the logistics company in each country, the tools needed to create, update and delete entities related with the network, namely Warehouses and Connections. System PC (Path Calculator) is responsible for calculating paths given a group of constraints.

The technological stack for System LNC is composed by a FrontEnd (WebApp) written in AngularJS (Google, n.d.), a BackEnd written in Java with Spring Framework (Pivotal Software, n.d.) and backed by PostgreSQL (PostgreSQL, n.d.) as its Relational Database Management System (RDBMS). System PC is a BackEnd software written in Java with Spring Framework and has an embedded database called Neo4J, added as a dependency. Since System PC is the targeted system to be replicated, a more detailed technology stack for it is presented in the next figure.

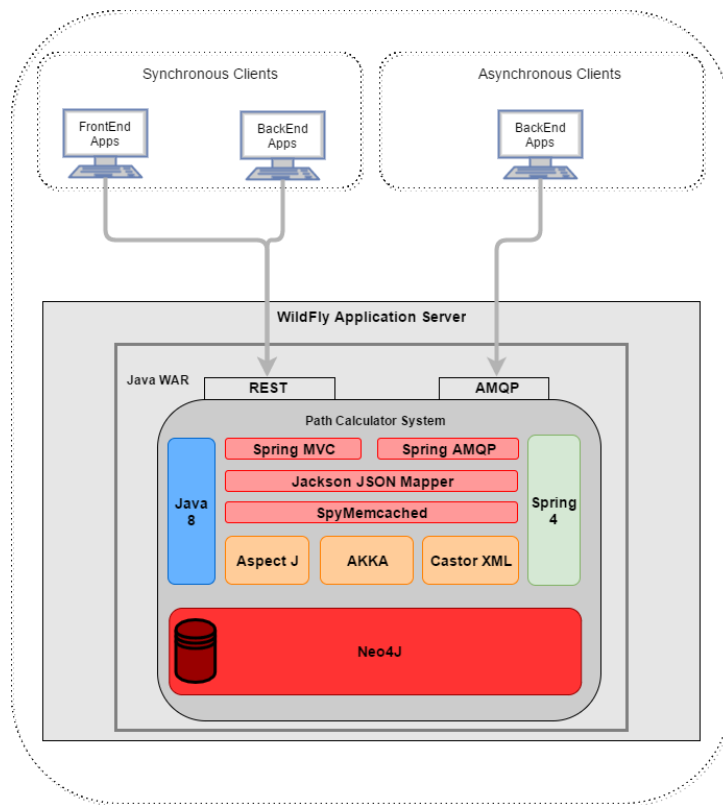


Figure 1 - System PC Technological Stack

Neo4J is a non-relational graph oriented database written in Java, and works around two main concepts, nodes and relationships. In the presented scenario, a Node represents a Warehouse, and a Relationship represents a Connection. Nodes and relationships represent the main entities managed by System PC. Each relationship must connect at least two nodes, otherwise it can't be created. For now on, 'nodes' and 'relationships' will be used while describing the system behavior overview. Figure 2 – Logistics Support System High Level Architecture shows the high level architecture for the whole system.

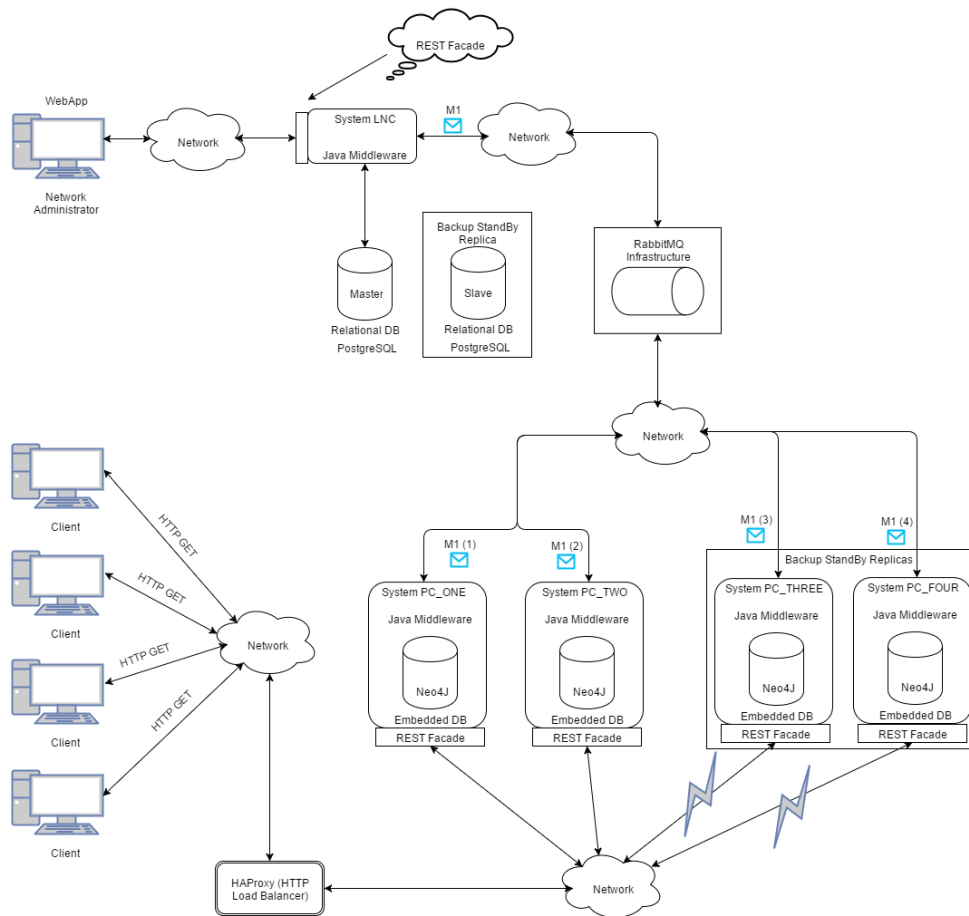


Figure 2 – Logistics Support System High Level Architecture

The network infrastructure is mutable, hence, it undergoes some changes over the time, like adding, changing or removing nodes or relationships. These actions are performed in System LNC by a logistics network administrator, or another user who has an equivalent access to the system. The network administrator interacts with System LNC via its frontend, built in AngularJS. The frontend communicates with the backend via Representational State Transfer (REST) compliant Hypertext Transfer Protocol (HTTP) endpoints.

When the network administrator creates a new Node, System LNC verifies if the business related constraints are correctly fulfilled and persists the new node on the master PostgreSQL database. There is a backup standby replica of the database which is in fact a slave, but this replication is managed by Amazon Relational Database Services (ARDS) (Amazon, n.d.) as the infrastructure is deployed on Amazon WebServices (AWS) (Amazon, n.d.). Replication being managed by ARDS not only incurs in additional costs as it is a payed service, but also implies not having full access the machine where the database is running.

As a change on the network is performed, when the change is persisted on the database of System LNC, that information must be propagated on the network for other interested systems. Particularly, in this case, System PC is interested in those changes, as it must have up-to-date network information to be able to provide fresh knowledge about the best Paths for a package to travel.

The communication between systems is performed via message queueing using RabbitMQ (Pivotal Software, n.d.) Message Broker. As such System LNC publishes a message for each change in the network where it happened. Interested systems can register a queue in RabbitMQ from where they want to listen for messages. The messages are multiplied by the number of queues subscribing them.

As described earlier, System PC is interested in being notified about changes in the network, hence it registers a queue in RabbitMQ infrastructure, where it will listen for those changes. When System PC receives a message, it performs the required validation and persists it on the Neo4J Graph database, providing new Path requests with up-to-date information about the logistics network infrastructure.

The diagram in Figure 2 – Logistics Support System High Level Architecture shows several replicated instances of System PC, namely PC\_ONE, PC\_TWO, PC\_THREE and PC\_FOUR. Each instance registers a listening queue on RabbitMQ infrastructure, which means, as the diagram shows, that when a message 'M1' is generated by a network change in System LNC, when it arrives to RabbitMQ message broker, it is multiplied by the four listening replicas of System PC.

The following Unified Modeling Language (UML) sequence diagram can ease the understanding of this flow.

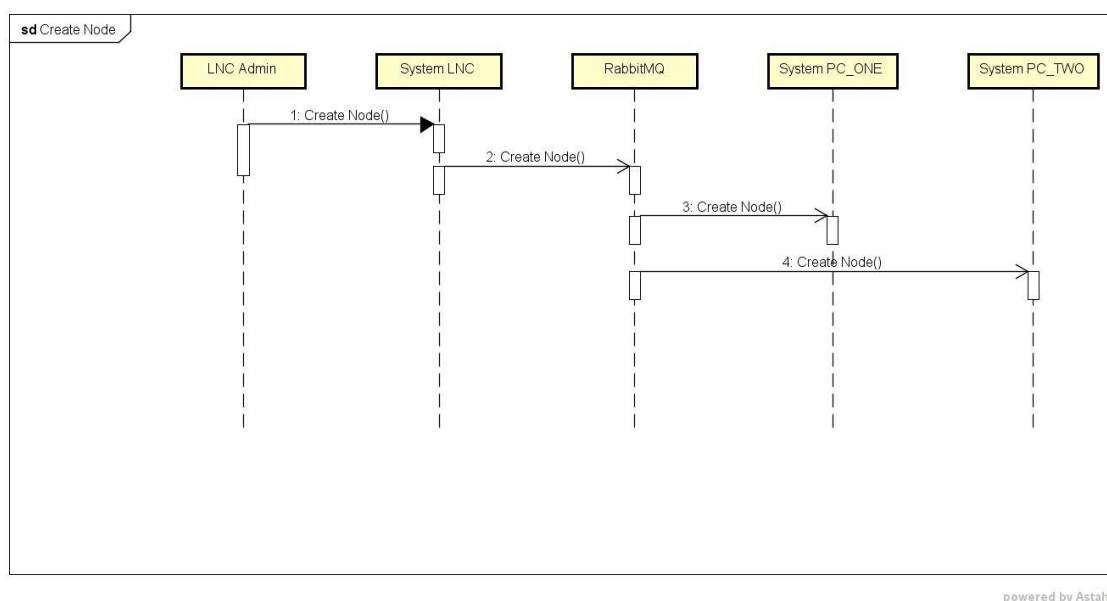


Figure 3 - Create Node Sequence Diagram

As this diagram helps to demonstrate, both System 'PC\_\*' are likely to receive a copy of the Create Node Message. However this is not guaranteed. As such, there is a considerable probability of ending up with the two instances unsynced.

System PC\_ONE and PC\_TWO are queried by clients for Paths calculation. These queries come in form of a GET HTTP request and in some cases, depending on the current topology of the logistics network and the required constraints, can be very work intensive operations. For this reason requests addressed to Systems PC are load balanced via HAProxy (HAProxy, n.d.). PC\_THREE AND PC\_FOUR Systems are backup standby replicas that can be swapped by PC\_ONE or PC\_TWO in case of failure. Client request queries are only of the read type, and at

the moment, they are about one thousand per hour but those number are likely to escalate as new countries adhere to the system.

System LNC is the owner of the information, as such changes on the network are always performed on System LNC, the main problem is that the Systems PC are relying on RabbitMQ for their replication, which does not guarantee that they are synced with each other. It is acceptable by the business that at some point of time they can be momentarily out of sync between them, but for instance if a problem or error consuming a message happens on PC\_TWO, this system will be permanently out of sync both with PC\_ONE and System LNC which is the truth owner. As of now, there is no way for PC\_TWO to automatically recover and sync itself with other PC\_ONE.

Overviewing the overall architecture, it is clear that there could be some issues with data consistency between systems. Not only between replicated PC systems but also between System LNC and Systems PC. It is worth noting that System PC only uses a subset of Node and Relationship entities information that is created on System LNC. In fact System LNC uses a relational database and it's data model is much more complex, containing other entities than the ones described here. Also as budget is a concern, this is a multi-tenant system, which means that the networks representing each country are not spread across specific instances per network, but instead they are all in each of System PC instances. This is true for both LNC and PC systems.

Entities created in System LNC are not guaranteed to be equally persisted on both System PC\_ONE and System PC\_TWO. As they are exactly two equal instances, none of them has the responsibility to coordinate the data with each other. Therefore if data difference is spotted between them there is no easy way to tell which one is not synced with System LNC. The resulting scenario is a manual debugging execution. For the current throughput this has been handled quickly by the DevOps team, but as new country networks are added, and the throughput rises, the debugging time will rise exponentially and any downtime will affect not one but the calculations for Paths on many country networks.

Also as the Logistics network grows, more replicas will be needed to be added to the infrastructure in order to distribute the request load coming from clients. It is important that consistency between those replicas can be maintained by a replication system. It is however not required by the business, that this replication should be performed atomically using distributed transactions. In fact, as can be stated by the current architecture, it is already following an Eventual Consistency approach, although the consistency convergence is not guaranteed right now.

The motivation for this work arises from the necessity to enable clustering capabilities at medium/long term to System PC, without resorting to any commercial solution in order to provide High Availability to the aforementioned system. The described system is a multi-tenant application that serves the logistics networks for 13 countries and is being periodically rolled out to new ones. As a new country is rolled out, more client applications need to be integrated. Currently the system is serving about 1000 request per hour, but with the current frequency of country additions to the system, this number is expected to double. Right now, the only way to provide High Availability for System PC is using Neo4J own Neo4J-HA solution, which imposes a prohibitive license fee for our current business strategy.



In order to solve this problem, the following pages will present an overview on database consistency and replication concepts. To conform to the current technological stack and following the company directions, the main restriction for the work presented in this document, is that it must use Java as its main programming language.

### **2.1.1 Functional Requirements**

#### **Requirement A:**

Provide a way to enable clustering on Path Calculator System without resorting to commercial Solutions.

A.1 - The system should remain usable if one of the System PC instances goes down;

A.2 - Failover between instances should be completely transparent to end users when one or more instance fails;

A.3 - There should be no lost transactions. Exception is only if the master instance fails and shuts down before broadcasting them.

#### **Requirement B:**

The framework should be flexible enough to be applied on applications backed by other database technologies.

#### **Requirement C:**

It must be developed in Java to follow the company established technology stack.

### **2.1.2 Non Functional Requirements**

- As System PC is very much geared towards read operations, a Master-Slave replication scheme should be used;
- There should not be distributed transactions within the cluster nor any type of distributed transaction locks;
- There should not exist any difference between the Master and Slave instances, apart from the current role they play in the cluster;
- Every service instance can assume the role of the Master in case of failure of the Master instance.
- Read load is balanced by an external load balancer and can be addressed to any replica.

- If one of the clustered instances fails, when it comes up, it should be synchronized with the remainder cluster.
- It is acceptable, if within the next 10 minutes after a transaction is committed on the Master instance, divergent data is observed on some slave instances.



## 3 State of the Art

### 3.1 Background Concepts

#### 3.1.1 High Availability

High Availability is a term used to quantify the availability of a system, which specifies that mission critical systems should have at least 99.999% availability (Gray & Siewiorek, 1991). The 'Availability' term by itself represents the time that a system is working properly (Marcus & Stern, 2003), and can be measured by the following equation that takes into account the Mean time to failure (MTTF) and Mean time to repair (MTTR) (Cecchet et al., 2008; Liu & Zsu, 2009; Marcus & Stern, 2003; Ozsu, 2007; Tanenbaum & Van Steen, 2002):

$$Availability = \frac{MTTF}{MTTF + MTTR} \quad (1)$$

The result of the equation is usually expressed as a percentage which in the end, translates into downtime in minutes. To achieve higher availability, systems have to be more resilient to failures and lower their downtimes. The following table is used to classify the different degrees of a system availability (Gray & Siewiorek, 1991; Marcus & Stern, 2003):

Table 1 - Availability Categories (Gray & Siewiorek, 1991)

Category	Availability	Downtime per year	Downtime per week
Managed	99%	3 days 15 hours and 36 minutes	1 hour and 41 minutes
Well-Managed	99.9%	8 hours and 45 minutes	10 minutes and 5 seconds
Fault-Tolerant	99.99%	52 minutes and 30 seconds	1 minute
High-Availability	99.999%	5 minutes and 15 seconds	6 seconds

To achieve High Availability, single points of failure (SPOF) must be eliminated (Cecchet et al., 2008). One way to achieve this is to use Clustering and Replication (Tanenbaum & Van Steen, 2002). Replication can be easily achieved with stateless applications. Several copies of an application can be deployed, forming an application cluster. By replicating applications, both scalability and availability are increased. Scalability can be increased by load balancing requests between application instances, and if one instance goes offline, other instances ensure that the system remain available. However, when data persistence is involved, replication is not as simple as deploy several instances of the application, replicating

databases is a challenging task and has been subject of many studies (Daudjee & Salem, 2006; Gray et al., 1996; Helland, 2007; Saito & Shapiro, 2005).

As replication is a key concept to achieve high availability, and the applications in the context of this master thesis rely on database systems for data persistence, the following chapters will overview important database concepts which will serve to support the replication strategies chosen.

### 3.1.2 Database Consistency

Data consistency has always been a concern when working with databases. Even within a single database instance, concurrent transactions could lead to data inconsistency if they access the same data. The problems arising from data inconsistency on concurrent transactions were defined by ANSI SQL-92 standard as phenomena (ANSI X3.135-1992, 1992; Berenson et al., 1995). To solve phenomena, ANSI specified transaction Isolation levels. Both phenomena and Isolation levels have been widely studied and are well documented (Berenson et al., 1995; Liu & Zsu, 2009; Ozsu, 2007). There were three main phenomena identified by ANSI SQL-92:

- **P1 – Dirty Read:** A Dirty Read happens when a transaction T2 reads a value that was modified by a not yet committed or rolled back transaction T1. If T1 performs a rollback, T2 will hold a value that physically never existed in the database.
- **P2 – Non-repeatable read:** This phenomena represents the case when a data item is read two times in a transaction T1, and between the two T1 reads, a transaction T2 modifies that data item. T1 will end up working with different values for the same data item.
- **P3 – Phantom:** If a transaction T1 executes two times a predicate that returns a set of rows, and between these two executions a transaction T2 inserts, modify or delete data that satisfy the predicate used in T1, then T1 will end up with two different data set for the same predicate. This phenomena, is similar to P2, but instead of referring a single data item, it refers to a data set.

To address these three phenomena, four transaction Isolation levels have been initially specified by ANSI SQL-92:

- **L1 – Read Uncommitted:** Transactions with this isolation level are prone to be affected by the three phenomena identified. In this isolation level, a transaction can successfully read and work with uncommitted data insertions, modifications and deletions made by other transactions.
- **L2 – Read Committed:** In this transaction isolation level dirty reads (P1) are not allowed, but non-repeatable reads (P2) and phantoms (P3) are still possible. A transaction T1 can only read committed data. Nevertheless if it reads a data item two times at separate moments, and between them, a transaction T2 modifies the same data item and commit, T1 will end up having two values for the same data item. The same is verified for phantoms (P3).

- **L3 – Repeatable Read:** With this isolation level, only phantoms (P3) are allowed. The idea is to ensure that transaction executions are serializable. This can be achieved by applying locks on both read and writes at a row level.
- **L4 – Serializable:** None of the referenced phenomena are allowed in this isolation level. To achieve this, transactions should be fully serializable.

Although ANSI SQL-92 defined L4 as Serializable, a fully Serializable Isolation level should enforce a fully serial execution between transactions, hence L4 isolation level is sometimes called “Anomaly Serializable” instead of the less granular, more broad term “Serializable” (Berenson et al., 1995; Ozsu, 2007).

### 3.1.2.1 One-copy Serializability (1SR)

1SR was the original de-facto standard of correctness criteria for data replication and consistency. 1SR states that the effect produced by running global scoped distributed transactions between several database instances, should be equivalent to executing the same transactions serially on only one database instance (Cecchet et al., 2008; Liu & Zsu, 2009; Ozsu, 2007).

Although 1SR was a very strong correctness criteria, it induced severe performance penalties as it prevents concurrent accesses to the same data items using locks. Not only transactions have to wait for each other to finish when accessing the same data, but also, deadlocks can occur. For instance, given two transactions, transaction T1 wants to access data X and then access data Y, while the other transaction T2 wants to access data Y and then data X, something like the following scenario can happen:

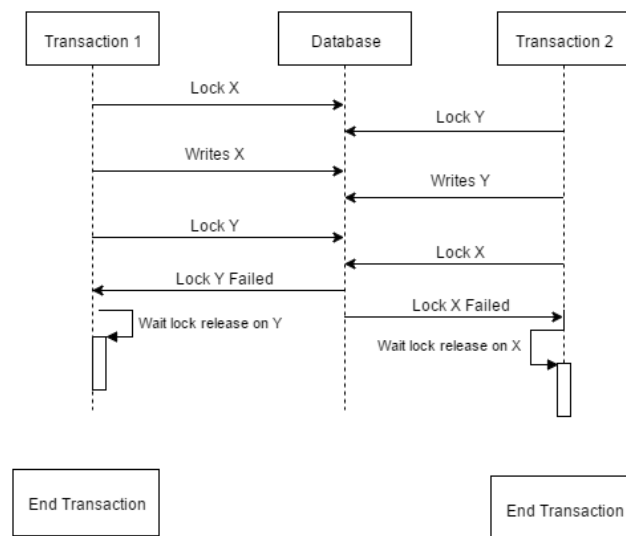


Figure 4 - Transaction Deadlock UML sequence diagram (Liu & Zsu, 2009)

This scenario ended up with a deadlock because the modified data only becomes unlocked when the correspondent transaction commits, but neither one will commit until they update all the data items in the execution plan. These drawbacks lead to the appearance of weaker transaction isolation levels like Snapshot Isolation (SI).

### 3.1.2.2 Snapshot Isolation

Proposed by (Berenson et al., 1995), Snapshot Isolation is a type of Multi Version Concurrency Control (MVCC) and aims to provide a more relaxed transaction isolation level to avoid the performance penalties present in 1SR (Gray et al., 1996).

In SI, each transaction  $T(n)$  reads a snapshot of the most up to date data committed in the database at the timestamp the transaction started (TS). When started, a transaction  $T1$  will never be blocked from reading, even if other transaction,  $T2$ , is dirtying the data items  $T1$  is reading. Each  $T(n)$  runs with their snapshot of data, and therefore  $T1$  will never see the changes of  $T2$  as long as  $T2$  has a superior TS. Also updates, inserts and deletes within  $T1$  will be reflected in  $T1$  snapshot, this way  $T1$  can work with its updated fields if it accesses them again. When  $T1$  is about to commit, it gets a Commit Timestamp (CT). If no other transaction has committed data that  $T1$  updated, in  $T1$ 's  $[TS - TC]$  time interval, then  $T1$  commits, otherwise it rolls back. (Berenson et al., 1995; Ozsu, 2007).

### 3.1.2.3 Atomicity, Consistency, Isolation and Durability (ACID)

ACID (Haerder & Reuter, 1983) is an acronym for what Gray defined as the basilar properties for a reliable and consistent transaction management system (Gray, 1978). The four properties are defined as follow:

- **Atomicity:** Atomicity aims to ensure that all actions performed within a transaction context should be treated as a unique action and either all will commit, or rollback in an 'all or nothing' approach. For example, if in the same transaction two 'updates' and a 'create' are performed, if one of them fails, the others are reverted. A transaction only successfully commits if all actions are performed successfully (Gray, 1978; Haerder & Reuter, 1983; Ozsu, 2007, pp.344-45).
- **Consistency:** This property ensures that a transaction does not commit invalid data, either by reading or writing dirty (uncommitted) data belonging to other transaction. If a transaction sees dirty data from other transactions, this can lead to an inconsistent state, if for example the transaction from which the uncommitted data belong rolls back (Gray et al., 1976; Gray, 1978; Haerder & Reuter, 1983; Ozsu, 2007). Consistency is tightly related with the next property, Isolation.
- **Isolation:** Isolation defines that a transaction  $T$  should see only the consistent state of the database within its execution context. That means that the transaction  $T$  should not see uncommitted data from other transactions and also other transactions should not see uncommitted data from transaction  $T$  (Gray, 1978; Haerder & Reuter, 1983; Ozsu, 2007). As seen previously, this property is essential to ensure Consistency.
- **Durability:** Durability ensures that any changes made to a database after a transaction is committed, should be permanently persisted, even in an advent of a system malfunction or error (Haerder & Reuter, 1983; Ozsu, 2007).

#### 3.1.2.4 Consistency, Availability and Partition Tolerance (CAP)

The CAP theorem (Fox & Brewer, 1999) states that distributed system with shared data can only guarantee two of the following properties:

- **Consistency (C):** Meaning having a single up to date copy of data (Brewer, 2012);
- **Availability (A):** Redundancy of data availability, meaning that the required data is always available and provided by one of the available replicas to the requesting client (Fox & Brewer, 1999);
- **Partition Tolerance (P):** A partition happens when for some reason there is a communication failure between two networked services. Being partition tolerant means that a system can continue to operate normally in an advent of a network node failure between replicas (Fox & Brewer, 1999).

(Fox & Brewer, 1999) Also states that “The stronger the guarantees made about any two of strong consistency, high availability, or resilience to partitions, the weaker the guarantees that can be made about the third”.

The importance of the CAP theorem grows as the need to scale a system increases. For instance if the system runs in one unique system with low access rates, it's almost guaranteed that all CAP properties can be achieved, as there is only one database, which is accessed by the backend which is on the same machine. Once the transaction rates increase and the system is horizontally scaled, CAP theorem becomes clearer. Horizontal scaling has its foundations on data partition, therefore, as P is already implied, tradeoffs must be made between C and A (Pritchett, 2008).

#### 3.1.2.5 Basically Available, Soft state, Eventual Consistency (BASE)

Basically Available, Soft state Eventual consistency (BASE) (Fox et al., 1997) principle is as an alternative to ACID, when Consistency is traded off by Availability. As BASE is a more relaxed principle that ACID concerning Consistency, higher availability is easier achievable, providing much higher levels of scalability than with ACID (Pritchett, 2008).

- **Basically Available:** Ensures that the system continues to respond when solicited, even if not with its full feature set. One case is when a system is composed by databases functioning in sharding. Sharding is a technique to distribute database records by several database instances. In this context, Basically Available could mean that if one of the databases instances is down, the system can continue to respond, although not with its full data (Fox et al., 1997; Pritchett, 2008).
- **Soft state:** Soft state means that the state of the system data can change over time regardless of user's inputs or actions. For example using a message queue system, data can be enriched by actions coming from one or more systems. If data is only partially complete, and the message broker goes down, the system ends up with incomplete data that will eventually be updated as soon as the message broker is online again. Stale or outdated data can be included in this scenario too. If a database is replicated by several instances, soft state allows that not all instances get updated



at the same time, and as a result ending up with stale data for some time until the update is executed (Fox et al., 1997; Pritchett, 2008).

- **Eventual Consistency:** Eventual Consistency is somewhat described by the last example on Soft State. It allows temporary inconsistency regarding data within the whole system, as long that the data eventually become consistent across the system after an acceptable period of time (Fox et al., 1997).

### 3.1.3 Database Replication

Database replication is a technique used to copy and maintain data within a database to other database instances, either in local and/or remote locations, obtaining what is often called a Distributed Database Systems (DDBS). The main motivations for building a DDBS are usually two, performance and availability improvement (Cecchet et al., 2008). Depending on the application, there are techniques that can be used to improve Read performance, or Write performance.

There are a bunch of different architectures for achieving database replication, Master-Slave also known as Single-Master, or Master-Master also known as Multi-Master are the most widely used (Cecchet et al., 2008; Liu & Zsu, 2009; Gray et al., 1996). For the sake of consistency, from now on in this document only Master-Slave and Master-Master terms will be used.

#### 3.1.3.1 Master-Slave Architecture

In this type of DDBS all the CUD operations are centralized at a single database instance, called the Master. It is then the responsibility of the Master to ensure the proper propagation of the CUD actions to the other database instances, called Slaves. This kind of architecture is easier to implement, as there is only one instance with the data 'ownership'.

Advantages (Cecchet et al., 2008; Liu & Zsu, 2009; Gray et al., 1996):

- As CUD operations are always performed in the same database instance, there is no need to manage possible conflicts, regarding different states for the same dataset on distinct database instances.
- Read operations can be redirected to Slaves to avoid overloading the Master;
- Horizontal scalability for Reads. Just add more Slaves when needed;
- There are no locks on the Slave database instances when Master database instance is being updated. (Only true when using Lazy replication);
- Slaves can be subtracted or added as needed with none-to-little impact on the Master;

Disadvantages (Cecchet et al., 2008; Liu & Zsu, 2009; Gray et al., 1996):

- All the writes are centralized at a single database instance, and as a result, for application with heavy CUD loads this can be a bottleneck;
- Possible downtime when the Master goes down;
- Need to manage “Master role” promotion to other instance in case of a Master failure;
- While “Master role” is being transferred to other instance, there is a possibility for data loss.

### 3.1.3.2 Master-Master Architecture

With Master-Master architecture, all database instances are equal amongst them. Meaning that, every instance is capable of processing both Write and Reads. In theory Master-Master can scale horizontally both Writes and Reads operations, but in practice implementation of Master-Master architectures had proven both complex and poorly scalable.

Advantages (Cecchet et al., 2008; Liu & Zsu, 2009; Gray et al., 1996):

- CUD operations can be addressed by any of the available database instances;
- Reads can be distributed by all instances;
- If one instance fails, the CUD request can be redirected to any of the other instances.

Disadvantages (Cecchet et al., 2008; Liu & Zsu, 2009; Gray et al., 1996):

- If the same data is updated in more than one instance, they must agree in the order of the transactions to be committed on each of them.
- As the volume of CUD operations grows, the more transactions will be needed to be dealt with by each one of the instances. Each instance will need to process every update, either if they are the initially targeted instance or not, as all transactions are propagated to the remaining instances. This will led to scalability problems, to the point that adding more instances will not bring more throughput (Cecchet et al., 2008).
- More complex implementation than Master-Slave architecture;
- Prone to transaction conflicts between database instances.

Disregarding the type of architecture used, the way to propagate a CUD operation through several database instances has also a big impact on the overall performance.

Traditionally, when building DDBS there has been an effort to maintain a unique consistency copy, meaning that for any given time, all database instances would be 100% synchronized (Saito & Shapiro, 2005). This kind of technique is called Pessimistic/Eager Replication (Gray et al., 1996; Ozsu, 2007).

### 3.1.3.3 Pessimistic Replication

When using this replication technique, all CUD operations received by the Master instance must be synchronously propagated (multicast) to the available database instances in a single transaction context to ensure atomicity. After the operation is multicasted, it must be successfully accepted on all the replicas to be effectively committed, otherwise it gets rolled back, meaning that it will end up with locking all database instances records or respective tables until the Global (Distributed) Transaction is committed. To achieve this, protocols like Two-Phase Commit (2PC) (Gray, 1978) are used to achieve atomicity within the distributed transaction. Using 2PC involves two phases, since the transaction is started until it ends. The first stage is the 'prepare phase', where the instance responsible for the initiating transaction gathers the commitment of the other instances on either commit or roll back the transaction. If they all agree on commit, the second phase is initiated. This step is where the commit is performed, this is called 'commit phase'. In 'commit phase', the originating instance coordinates with the remaining instances to effectively perform the commit. If the commit happens to be not possible, all instances are instructed to roll back (Bernstein et al., 1987; Ozsu, 2007).

This technique can work well on local networked environments, but within Wide Area Networks (WAN), where higher latencies, and not always connected systems are present, pessimistic replications leads to serious performance problems and deadlocks (Gray et al., 1996) (Saito & Shapiro, 2005).

Advantages (Gray et al., 1996; Liu & Zsu, 2009; Ozsu, 2007):

- Ensures data consistency across all instances, as replication between instances is synchronous. Uses a Global Distributed Transaction to either commit to all instances or perform an rollback on all instances;
- Maintains One-copy serializability (1SR).

Disadvantages (Gray et al., 1996; Liu & Zsu, 2009; Ozsu, 2007):

- Performance degradation increases with the added number of instances as the global transaction size grows;
- As the number of instances increases, also increases the deadlock probability;
- If any of the database instances is disconnected, the transaction fails.

Another technique is Optimistic/Lazy Replication (Gray et al., 1996; Liu & Zsu, 2009; Ozsu, 2007; Saito & Shapiro, 2005). This replication technique defines that it is acceptable, if for some short period of time divergent data is observed on some database instances.

### 3.1.3.4 Optimistic Replication

When using optimistic replication, the Master replica processes the CUD operation in a local scoped transaction, then it propagates the operation to the other database instances. Each database instance processes the operation in its own local transaction.

As CUD operations are propagated asynchronously, there are no “global locks to records or tables” amongst all database instances as in pessimistic replication, hence improving availability. However, this technique does not fully ensure all the ACID principles, more precisely Atomicity and Consistency.

Optimistic replication does not try to enforce full consistent data across database instance at all times, treating the synchronization asynchronously in the background and therefore, it is acceptable if for some short period of time, divergent data is observed between some instances.

Advantages (Liu & Zsu, 2009; Saito & Shapiro, 2005):

- No access block to database instances other than the one where the update is being performed. As there is no synchronized CUD propagation within a single global distributed transaction between instances, even if some instances are still performing their local transactions, others, either updated or not, are still accessible, hence improving availability;
- If a database instance is offline, it can be updated at a later time, when it comes up;
- Good horizontal scalability as there is no synchronous CUD propagation between database instances.

Disadvantages (Gray et al., 1996):

- CUD operations are not atomically executed in all database instances;
- Some inconsistent data can be observed between database instances within a short time period;
- Transaction conflicts when using Master-Master can be very complex to solve.

## 3.2 Existing Frameworks

Database replication is not always performed at the database layer. Actually, it is becoming more common to have middleware software taking care of those tasks. This section serves the purpose of studying what has been done in the field of replication at the Middleware layer. To narrow the scope to better suit the current needs, this chapter will focus only on solutions that were written in Java.

### 3.2.1 C-JDBC

Clustered JDBC (C-JBDC) (Cecchet et al., 2004) is an open source middleware based replication framework that allows heterogenic relational database clustering.

C-JBDC acts as a proxy between the application clients and the replicated databases. The client applications that are built around JDBC only need to replace the JDBC driver with the C-

JDBC driver that exposes the same interface. Client applications access to a virtual database as they were accessing to a single database.

The acronyms for the types of replication supported by C-JDBC are borrowed from Redundant Array of Inexpensive Disks (RAID). The C-JDBC team borrowed the RAID acronym and coined the Redundant Array of Inexpensive Database (RAIDb) term. As with conventional RAID, with RAIDb there are also several configurations. Each RAIDb configuration represents a database replication strategy or policy. The following replication strategies can be used with C-JDBC:

- **RAIDb-0 Table Partitioning:** Database tables are distributed along the several database instances. In this mode, it is required at least two database backends. Although it offers a small increase in performance, this mode does not ensure fault tolerance, as if one database fails, integrity is lost.
- **RAIDb-1 Full Replication:** This mode fully replicates the database schema and data across all database backends. It offers the most efficient fault tolerance level of the supported replication models. However, as Write operations must be broadcasted to all databases, the final outcome takes longer and therefore performance takes a hit.
- **RAIDb-2 Mixed replication:** This mode merges RAIDb-0 and RAIDb-1, meaning that it provides a hybrid replication. The main characteristic in this mode is that all database tables must be replicated at least in two database backends.

It is important to note that C-JDBC does not use 2PC protocols and distributed transactions to ensure consistency between the databases, instead transactions are run in parallel; each database backend gets its own transaction. If one transaction fails, the respective database backend is taken offline, rebuilt and synced with the Master. When it is again consistent with the cluster, it is reintegrated. Also worth point out, is that distributed joins are not supported, which means that all the data needed for a query must be present in the same instance. This will limit a great number of applications to use RAIDb-1.

When a client application makes a database call, the C-JDBC driver then calls a component named C-JDBC Controller that is responsible to act as a proxy between the C-JDBC driver and the database instances. Figure 5 - C-JDBC component architecture illustrates the overall component architecture.

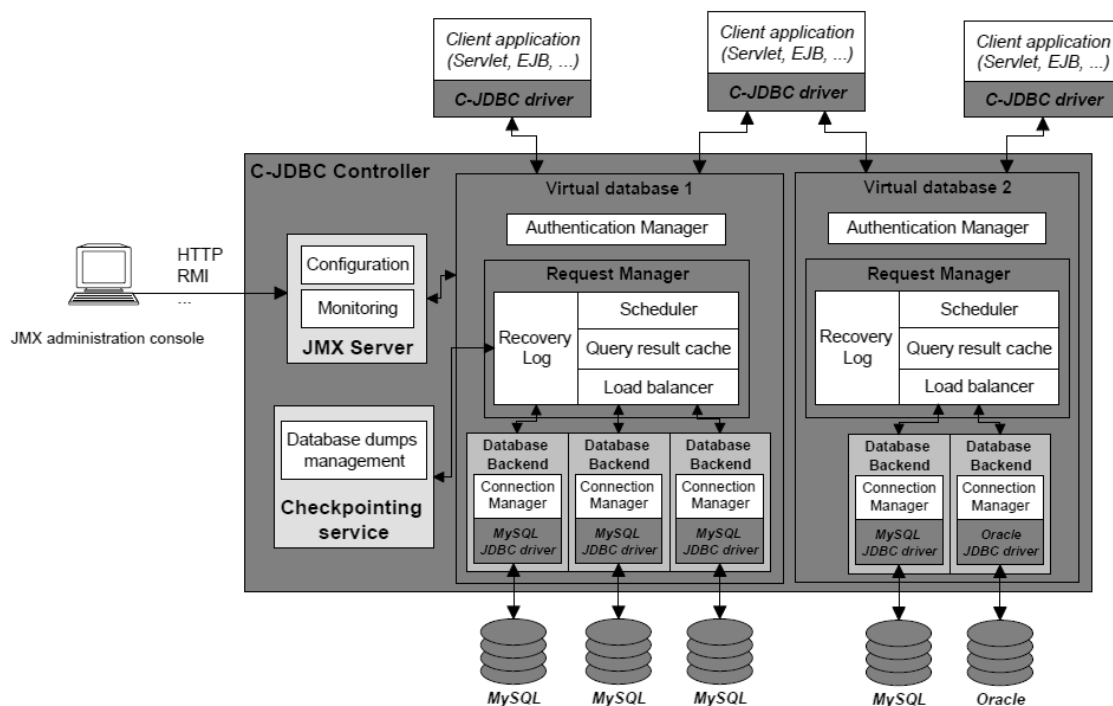


Figure 5 - C-JDBC component architecture (Cecchet et al., 2004)

C-JDBC controller itself is composed of several components, being the central one the Request Manager. The Request manager is always composed by a scheduler and a load balancer. The Recovery Log and Query Result Cache components are optional.

The Scheduler is responsible to distribute the transactions among the several databases. Read transactions are redirected to a single backend whereas updates are sent to one or more databases, depending on the replication policy. After a Write is issued, the response to the clients is always synchronous, and by default is only sent after all databases have responded to the transaction. However C-JDBC has an “Early Response” option, which means that in the case of a Write operation, the client application will get the response as soon as the first database returns an answer.

The Query Result Cache is used to return data for repeated queries. By default the cache is invalidated for data that has been updated on the database. Nevertheless the user can customize this setting opting to weaken cache invalidation. Obviously, weakening cache invalidation would present some inconsistency problems.

An important component in C-JDBC Controller is the Load Balancer. The Load Balancer is reached if the cache has not been hit for a previous executed query. The load balancer behavior and performance depends on the replication policy defined. If a RAIDb-1 type replication is used, then the load balancer can redirect the query to any database backend. If however, RAIDb-0 or RAIDb-2 is used, additional computation is required, as the load balancer needs to know the database schema present at each database backend to fulfil the query, and possibly orchestrate the query targeting at more than one database backend.

The recovery log, contains entries for every SQL action and the respective transaction identification. The recovery log can be used, for example, to recover database backends that have been put offline due to transaction failures or a Checkpoint (more on that below).

C-JDBC also provides a checkpoint feature. The checkpoint takes a snapshot from a database backend for latter recovery. During the checkpoint process the targeted backend is taken offline and the snapshot begins. The Recovery log, logs the start and the end of the checkpoint process. When the process is complete, the database backend is taken back online and the transactions registered on the recovery log during the checkpoint processing time, are replicated to the database backend. This process uses a somewhat Extraction Transform and Load (ETL) approach.

Another feature contained in C-JDBC is Java Management Extension (JMX) support. The JMX support allows for runtime monitoring and configuration using an administration console like JConsole bundled with the Java Development Kit (JDK).

To sum up, in the paper C-JDBC is a highly capable clustering framework. It offers support to vertical and horizontal scalability and even a mix in between the two. It uses Lazy replication (Optimistic) as the main replication technique, which improves both availability and performance, at the cost of some temporal consistence. Distributed Joins are not supported, so other replication schemes than RAIDB-1 will impose a big constraint on the types of queries supported. Unfortunately C-JDBC only works with relational databases, which makes it impracticable to use with Non Structured Query Language (NoSQL) databases like MongoDB and Neo4J.

### **3.2.2 Ganymed**

Ganymed (Plattner & Alonso, 2004) is a middleware replication system written in Java that provides lazy replication across a set of replicated databases. Its base principle is to clearly separate update from read transactions, using a scheduling algorithm called Replicated Snapshot Isolation with Primary Copy (RSI-PC), that works on a set of databases configured for Snapshot Isolation. Unlike C-JDBC, Ganymed works only with fully replicated databases. Below is the generic architecture when integrating applications with Ganymed.

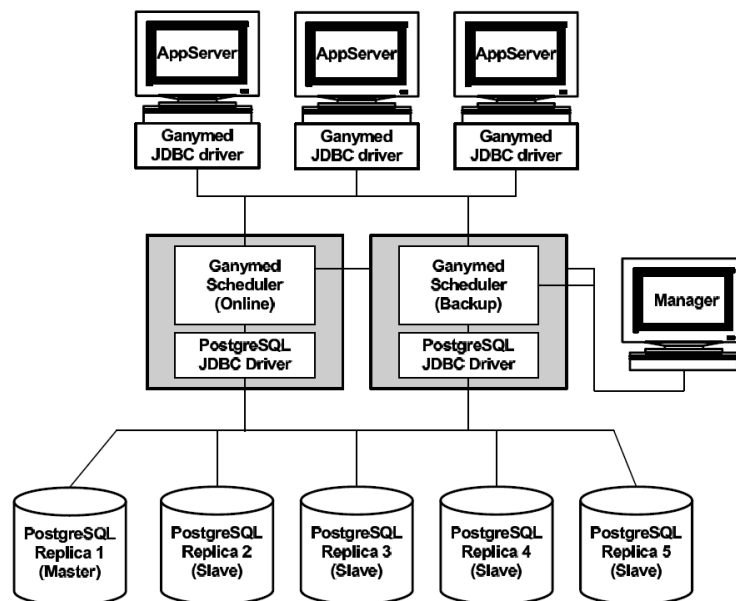


Figure 6 - Ganymed component architecture (Plattner & Alonso, 2004)

One of the main components for Ganymed is the Scheduler. The Scheduler uses the aforementioned RSI-PC algorithm to route queries between database instances. Client applications use a Customized JDBC driver to connect to the scheduler. In fact, this is the only change needed in the client side to integrate client applications with Ganymed. Only one working scheduler is supported, nevertheless a second scheduler can be configured for backup, and it will automatically replace the working scheduler in case of failure.

As already stated, the scheduler uses the RSI-PC scheduling algorithm. This is a lightweight algorithm which has the primary role of schedule transactions for a set of database instances. Queries go for separated instances whether they are read-only or CUD. Currently RSI-PC only supports databases using Snapshot Isolation, as Oracle or Postgres. In fact as far as database support goes, only those two RDBMS are currently supported.

RSI-PC imposes some constraints on isolation levels when executing a transaction in one of the supported databases. Read-only queries must be run in Serializable isolation level. CUD queries can run both in Serializable or Read Committed isolation levels. The isolation level at which each transaction must run is communicated to the RDBMS by the Ganymed Scheduler.

When the Scheduler performs a CUD transaction, it takes notice of the order of the commits within the transaction that has been performed on the master replica. It then replicates the write set, to each database replica, ensuring that the order of commits is exactly the same as it was in the master database. Also, with each transaction, is generated a global database version number. This global version number is used by the Scheduler to know which of the replicas had already been updated when choosing a database replica to fulfill a read-only query.

When using Ganymed, there are two types of roles a database replica can assume, Main, or Read-Only. At any time there can be only a main replica while there can be several read-only



replicas. RSI-PC always routes update queries to a master replica database instance, while reads are handled by the other replicas instances (Read-Only replicas).

Another important component within Ganymed architecture is the Manager. The Manager component is used by systems administrator not only for adding or removing replicas, but also to perform reconfigurations to the whole system, and to configure the policy for the scheduler substitution when a failure occurs.

Wrapping up, Ganymed is a replication middleware that does not provide the same level of flexibility as C-JDBC, but still, it can be an option to be used with Oracle and PostgreSQL backed systems. Apart from limiting to two relational database vendors, non-relational databases are also not supported.

### **3.2.3 Tashkent**

Tashkent (Elnikety et al., 2006) is a prototype solution built with the purpose of demonstrating that, the separation between transaction commits ordering, and their physically persistence to the disk imposes a significant bottleneck in scalability. In this context two different approaches were taken, one where durability is transferred from the database to the middleware layer, and another where global commit order is moved from the middleware to the database. The implementation for the first approach is called Tashkent-MW whereas the implementation of the second approach originated Tashkent-API. As this document is focused primarily in replication middleware, only Tashkent-MW is covered.

Tashkent-MW uses a multi-master approach. All database replicas are capable of performing both read-only and CUD operations. However unlike other multi-master architectures, Tashkent-MW does not rely on 2PC protocol to reach an agreement on the commit order of the transactions. Instead it relies on a core component called Certifier to mediate transaction commit order between database replicas.

There are three central groups of components in the Tashkent-MW architecture. The Database replicas, the Proxies and the Certifiers. The Certifiers play a central role in this architecture, as they not only act as intermediators for communication between the database replicas, but also certify and order the transaction commits between them. Client applications connect to the Proxy which is written in Java and exposes a JDBC interface. Internally it uses the correspondent JDBC driver for each database it connects.

Database replicas communicate with the exterior through their Proxy. Each database replica has their own proxy, and there is no direct communication between proxies. The Certifier is the component responsible to orchestrate all the information that is sent to the replicas.

The following figure illustrates the high level architecture of Tashkent-MW.

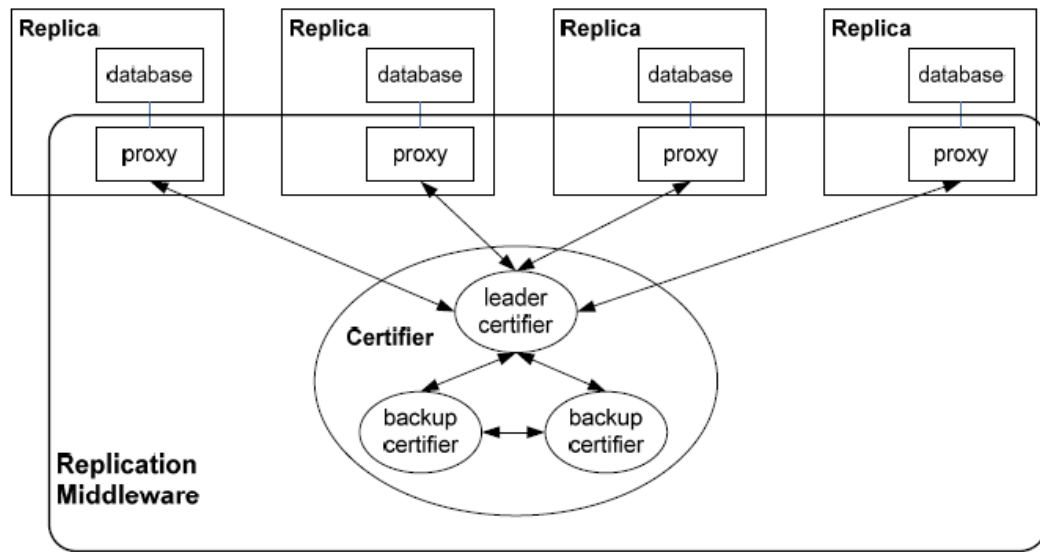


Figure 7 - Tashkent-MW component architecture (Elnikety et al., 2006)

As stated before, the Certifier is the component responsible to mediate transaction commits between the database replicas and is written in C. The main job of this component is to receive update request from the database replica proxies. The certifier maintains a global system version. Each successfully certified transaction increments the global system version and is written to a log. The log update is composed by the transaction write set, and the commit version, which is the same as the newly incremented global system version.

Update requests from proxies received at the Certifier consist of a series of transacted write sets, each one containing a start version. This start version is the global system version (provided by the certifier) that the correspondent replica was when it began the transaction. The Certifier then searches in the log for possible conflicting transactions that have a commit version higher than the start version of the transaction being certified. If there is no conflict, the global system version is incremented and the log updated. The Certifier then sends the 'commit' decision to the database replica, along with the commit version. The commit version is the same as the global system version that was incremented when the Certifier successfully approved the transaction. If a conflict is detected, the transaction is not certified and the decision to abort the transaction is sent to the database replica. Along with this decision, the Certifier also sends write sets committed by other database replicas (remote write sets) which have commit versions higher than the start version of the transaction requested for certification. The certification flow is illustrated in Figure 8 - Tashkent transaction certification flow

The Certifier component is replicated to provide availability. In this scheme, there is always a leader certifier responsible to receive and processes all the certification requests coming from the database replicas. The leader certifier then sends the new write sets approved to the remainder certifiers. The backup certifiers then process the changes and acknowledge them to the leader.

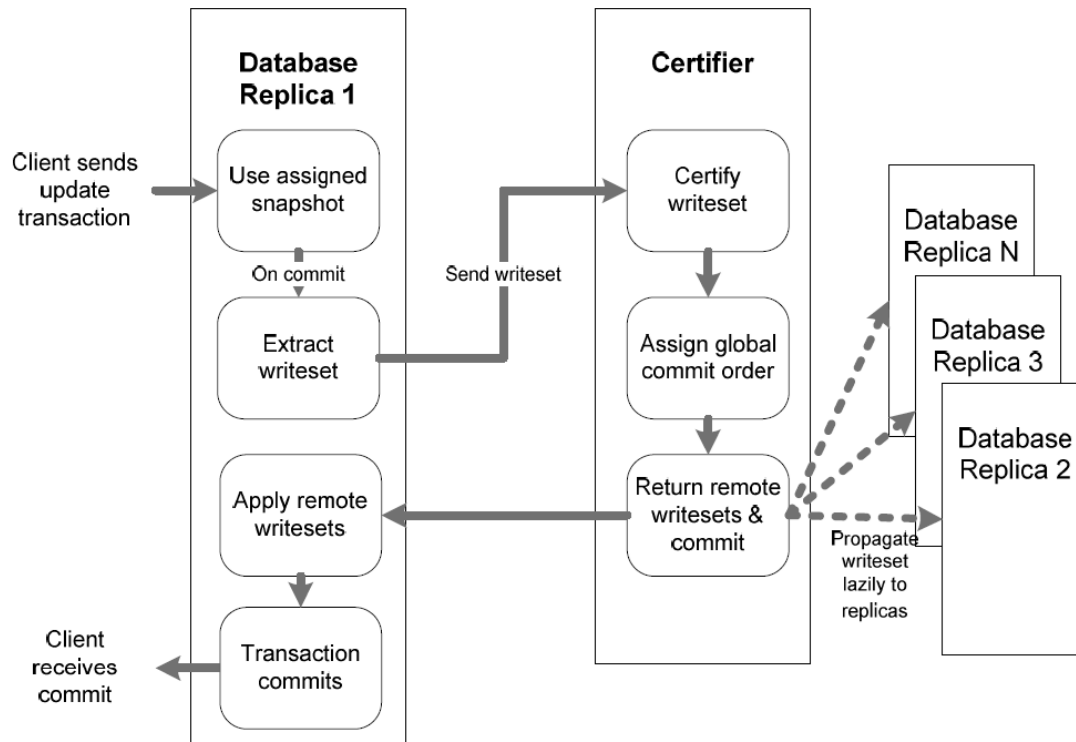


Figure 8 - Tashkent transaction certification flow (Elnikety et al., 2006)

The Proxy, is a component written in Java that acts as a transparent proxy to client applications, providing a familiar JDBC interface. On the database end, the Proxy uses the specific database driver for the database vendor.

Upon receiving a transaction from client applications, the Proxy first evaluates if it is a Read-Only transaction, and if it is not, it extracts the write sets and sends them to the Certifier where the flow described earlier is performed. If it is a Read-Only transaction the transaction is directly forward to the database.

Summarizing, Tashkent is a capable replication Middleware that combines transaction ordering and durability in the Middleware layer in one action. Groups of ordered transactions are written in blocks. This is similar to what happens in a single database system, avoiding having one disk write for each transaction. Although it is a viable option for relational databases, Tashkent do not support non-relational ones.

### 3.2.4 Neo4j High Availability

As stated earlier, one database technology that is used in the current technology stack is Neo4J, a Graph based NoSQL database. Neo4J offers a High Availability solution in its Enterprise edition (Neo4j-HA). Unfortunately there is a license fee associated.

From the client point of view Neo4J-HA may be seen as a multi-master solution. Write operations can be addressed to any database instance. However there is only one master. If a write operation is received by a slave instance, it coordinates this operation closely with the

master instance in the background. For this reason, it is always faster to address write operations directly to the master instance.

Each Neo4J instance includes a Cluster Management component. This component is responsible to keep track of every instance that join or leaves the cluster, electing a new master instance when necessary.

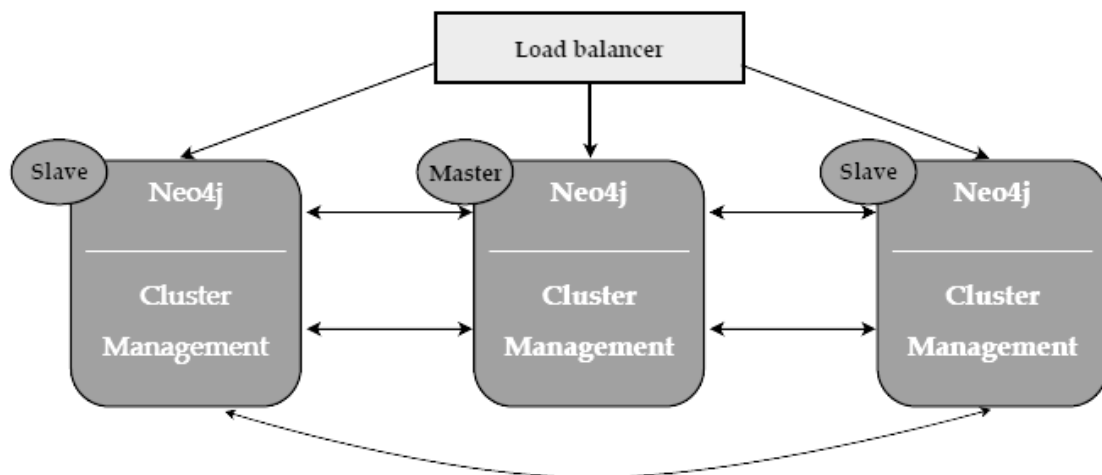


Figure 9 - Neo4J-HA component architecture (Montag, 2013)

The master is responsible to handle write operations. However for the master to accept such operation, more than half of the cluster instances should be available. In a scenario when only three or four of eight cluster instances are available, only read operations will be fulfilled.

Neo4J-HA does not support sharding, meaning that all instances should have the capacity to store the full replicated dataset. Although this may impose a performance penalty when restructuring indexes, read performance, can be increased adding more instances. However increasing the number of instances will only bring performance gains on multiple client access scenarios, and eventually every single instance performance will degrade as the data increases. Nevertheless Neo4J-HA uses the concept of 'Cache-based Sharding' meaning that client requests are always sent to the same database instance. This has the advantage of populating the cache with the graph surroundings of the requests from that client.

Wrapping up, Neo4J-HA is a good option when using Neo4J graph database, and would be a good viable option if not for its costly annual license fee.

### 3.2.5 Conclusion

This chapter reviewed some of the available clustering frameworks in the market, and while C-JDBC, Ganymed and Tashkent seem all pretty competent and complete solutions, they don't offer the flexibility to be applied to applications backed by non-relational persistence databases. Neo4J High Availability would suit the needs for replicating System PC, however it also does not support other databases than Neo4J, and its licensing price is too high. As such, a custom solution needs to be developed to fulfill both the Functional Requirements and Non Functional Requirements described in the Context chapter.



## 4 Proposed Solution

### 4.1 Introducing Replic8

Replic8 (R8) is a Java framework that can be added to any Java project as a library dependency, to enable data replication in an application cluster. As seen earlier, the target system to replicate, the System PC, is composed by an embedded NoSQL database written in Java. As such there is no remote communications between the middleware and the persistence layer. This architecture disables the possibility to use the same approaches in this framework, as the ones seen in State of the Art chapter, where the connection to the database containing the SQL commands, or datasets are intercepted by a custom database connector that acts as a transparent proxy and redirects them to the replication framework.

### 4.2 Replic8 High Level Transaction Replication Flow

In order to replicate transactions, all the application instances that form the cluster should be configured with Replic8. For each running instance, Replic8 should be either configured as master (only one), or as slave. Figure 10 shows an UML activity diagram representing the high level overview of the flow to replicate a transaction to a slave instance when a write operation is received on the master instance.

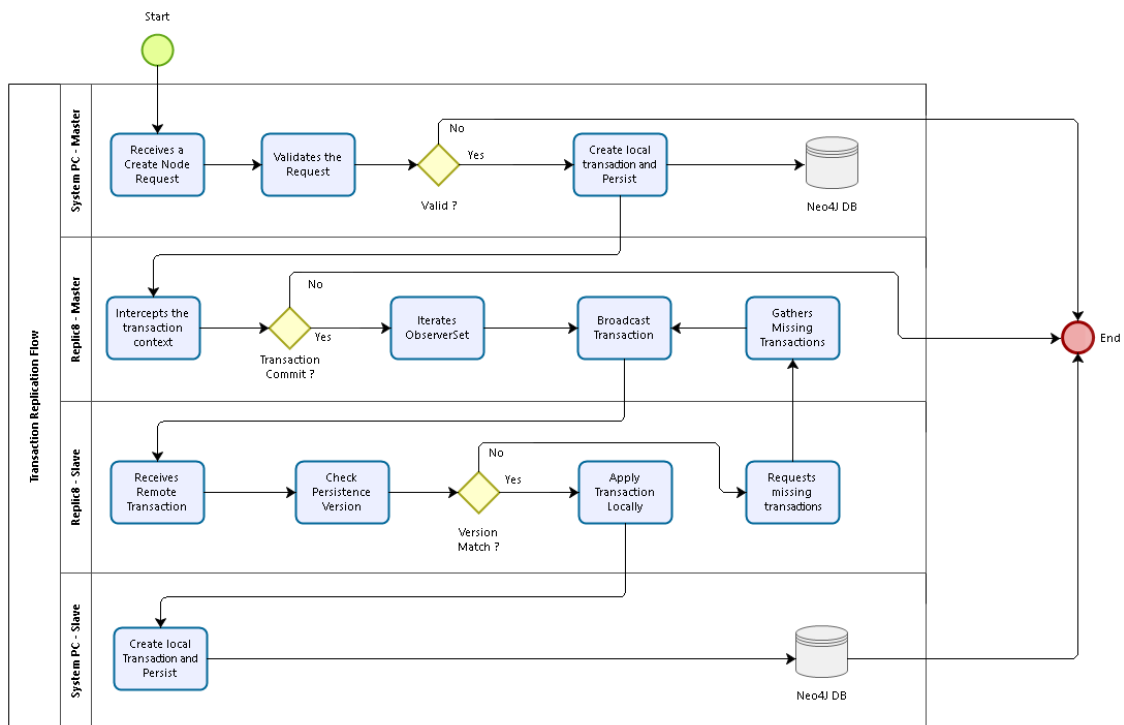


Figure 10 - High level transaction replication flow UML activity diagram

Replic8 is designed to intercept transactions in the middleware, targeting classes that orchestrate transacted persistence operations, like service or model classes.

When one of those classes is called, Replic8 intercepts its method call and evaluates the outcome. If the outcome is a commit, then Replic8 sends the transaction to the slave(s) instance(s). When slave instances receive a remote transaction (sent by Replic8 master instance), they check if the version of the remote transaction matches the next version in the local transaction version sequence, and apply the transaction locally. If there is a version mismatch, the slave informs the master, which in turn will respond with the missing transactions.

## 4.3 High level Architecture Overview

Replic8 is built around four core modules. The Cluster module is responsible to manage the cluster composition and assure persistence convergence within the cluster. Transaction module handles the main operations regarding transactions, such as transaction broadcast. Health Module plays a central role in actively monitoring each cluster instance state and maintains the cluster availability. The Recovery Logger module is responsible to maintain a history of each transaction that has occurred in each cluster instance.

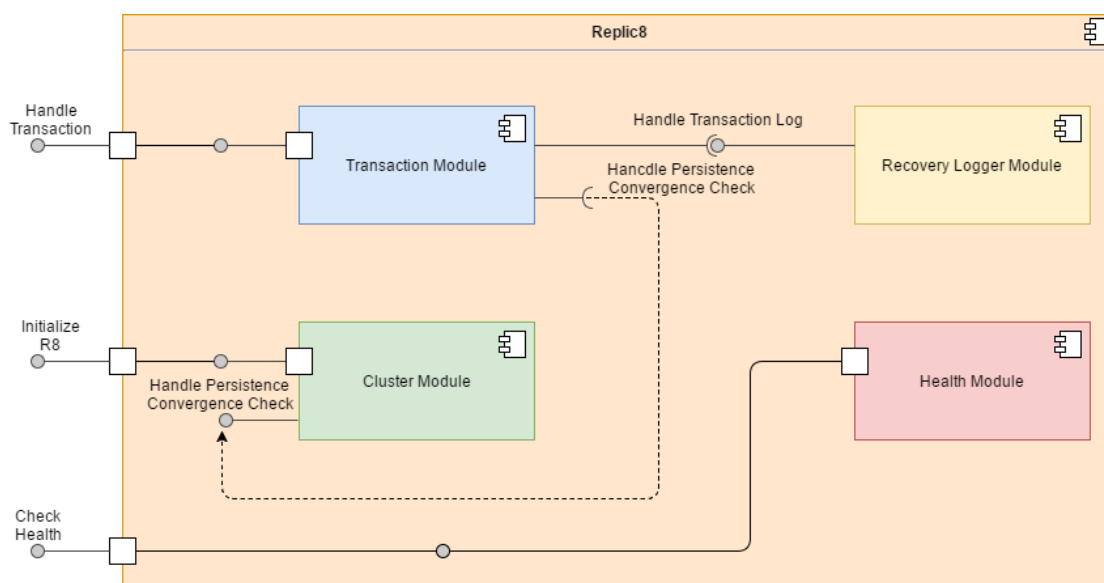


Figure 11 - Replic8 High level UML component diagram

### 4.3.1 Transaction Module

As stated earlier, the Transaction Module, is the module responsible to handle the processing regarding the transactions themselves. This module performs distinct actions depending on which role R8 is configured to. If R8 is configured as master, this module is responsible to handle transaction interception in the master instance of an application, and to broadcast it to slave instances. If R8 is configured as slave, this module is responsible to receive transactions

broadcasted by the master R8 instance and process them locally. Figure 12 shows a high level overview of the subcomponents contained in the Transaction Module.

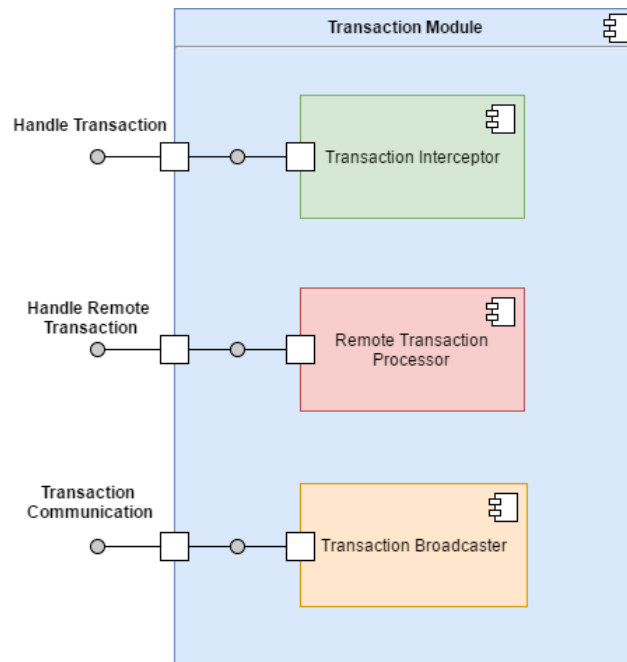


Figure 12 - Transaction Module UML component diagram

The Transaction Interceptor (TI) component works closely with application classes that employ the Service pattern or equivalent. Actually, the framework logic can be applied to lower level classes as the ones implementing the Data Access Object (DAO) pattern. However as DAO persistence actions can be rolled back if one or more DAO's in the same transaction context fail, this is not a reliable option. Choosing to intercept at the Service level seems the logic choice, as the result on these classes usually represent the result of the whole transaction. The TI component will be responsible to evaluate the return of each call to the aforementioned 'Service' classes methods and act accordingly. If the result of the called method translates in a transaction commit, then the component calls the Transaction Recovery Logger (TRL) component followed by the Transaction Broadcaster (TB). If not, it means that a rollback or persistence error has happened. As such, there is nothing to replicate, and the flow ends.

Transaction Broadcaster (TB) is the component is responsible to handle the transaction context broadcast between the master system and the slave replicas. As stated earlier, all successfully committed transactions are sent to the other slave replicas in the cluster, along with the PV generated. Transaction Broadcaster implements an Observer Patter, which means that each slave instance in the cluster registers itself as an observer for transactions. When a transaction is successful, TB broadcasts it to all the registered observers. TB assumes a sender role in the master instance, and a receiver role in the slave instances.

The Remote Transaction Processor (RTP) handles all the incoming remote transactions received by the TB when in slave mode, and applies them in the slave instance. Before



applying the remote transaction locally, it works in conjunction with the PVS to check if the updated PV version will match the one received with the remote transaction.

### 4.3.2 Recovery Logger Module

This module is responsible to maintain an ordered transaction history in each of the cluster instances. The log history persisted in the master instance is used to provide transaction information to out of sync slaves, being them long time running slaves, or a new slave machine just added to the cluster. The following figure shows an overview of the components that compose this module.

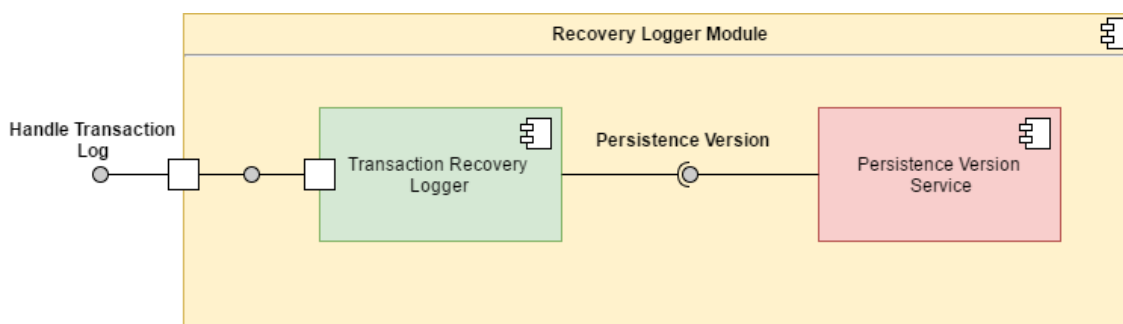


Figure 13 - Recovery Logger module UML component diagram

A database transaction log is maintained by the Transaction Recovery Logger (TRL) component. This component is responsible to log all the committed transactions both in the master system and in the slaves. This log can be used in an advent of a failure in the master system or in any of the replicas, to rebuild any instance that become out of sync.

Working closely with the TRL is the Persistence Version Service (PVS) component. For every committed and logged transaction, a new database Persistence Version (PV) is assigned. The version assigned for each transaction is incremented from the previous version following the  $nv = pv + 1$  equation, where  $nv$  is the new version, and  $pv$  is the previous version. This component is responsible to manage the persistence version for the transaction and communicate it to the TRL in order to maintain an ordered transaction history.

The PV is sent to the remote slave replicas along with the transaction context itself. This will ensure that the targeted slave replicas should converge to the same persistence version as the master replica. When applying a remote transaction, slave replicas will check if the internally generated PV matches with the one received along with the remote transaction. If not, they will respond back with a sync request to the master system, along with the last PV they have. The master will then, in turn, respond with the correspondent transactions for the sync requested PV onwards.

### 4.3.3 Cluster Module

The main responsibility of Cluster Module is to manage the cluster, registering slaves, and assuring persistence convergence within the cluster.

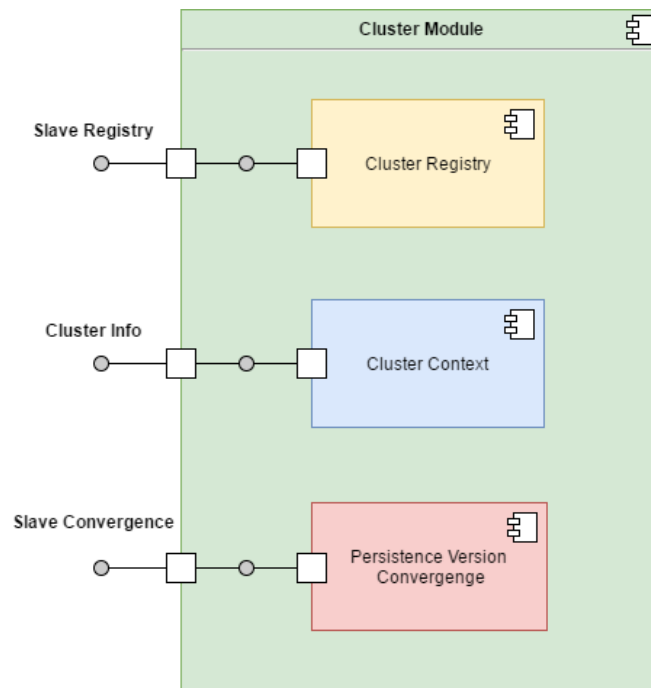


Figure 14 - Cluster Module UML component diagram

Cluster Registry (CR) is the component responsible to handle slave registrations in the master instance. Only registered instances will receive a copy of the transactions broadcasted by the TB. Slave registrations can happen both when the master starts for the first time and there are already passive slaves running (more on passive and active slaves in the next chapter), when a new active slave joins the cluster, or when a new master is elected.

The Cluster Context (CC) component maintains a catalog of network addresses for the cluster master and slave systems. The network address catalog is updated manually by the system administrator directly in the master system. Although only the master instance makes use of the full catalog of the network addresses for replication purposes, it also synchronizes this info with the remainder instances. This is helpful in an advent of a master failure, where another instance assumes the master role. Therefore, as all slaves receive the network addresses catalog of the full cluster, the new master instance do not need to be reconfigured by the system administrator with all the cluster network addresses.

The Persistence Version Convergence component (PVCC) ensures that all slaves eventually converge to the same version as the master. Every time a slave receives a broadcasted transaction, this component checks if the slave is currently at the immediately previous persistence version. If not the slave informs the master that it needs additional transactions.

#### 4.3.4 Health Module

The main role of the health module is to employ the 'Basic Available' part of the BASE acronym. This module is responsible to send and receive periodic checks from all instances in order to check their availability, and handle the failover in case of a master instance failure.

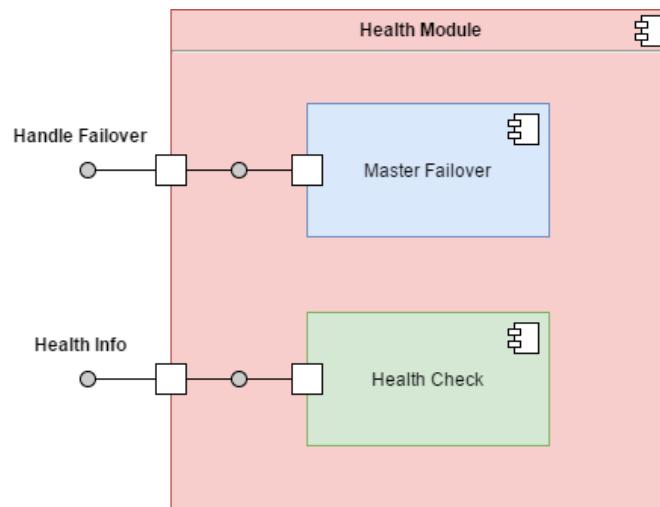


Figure 15 - Health Module UML component diagram

Master Failover component (MFC) is responsible to handle the transition of the master role from one instance to another, when the current master fails. The master transition hierarchy is defined in CC for each cluster instance by the administrator. As soon as the master role transitions to a slave instance, the previous master will be placed last in the hierarchy as soon as its instance rejoins the cluster.

Health Check component (HCC) periodically checks for the health of all instances in the cluster. As master, this component will check if all the instances are up and running. Instances that become offline are removed from TB observer list. When in slave mode, this component checks if the master is up. If not, the slave delegates further actions to the MFC component.

## 4.4 Design Approaches and Decisions

The most similar approach to the ones presented in State of the Art chapter would be constructing a wrapper around the Neo4J API that would intercept persistence operations and broadcast them to another instances.

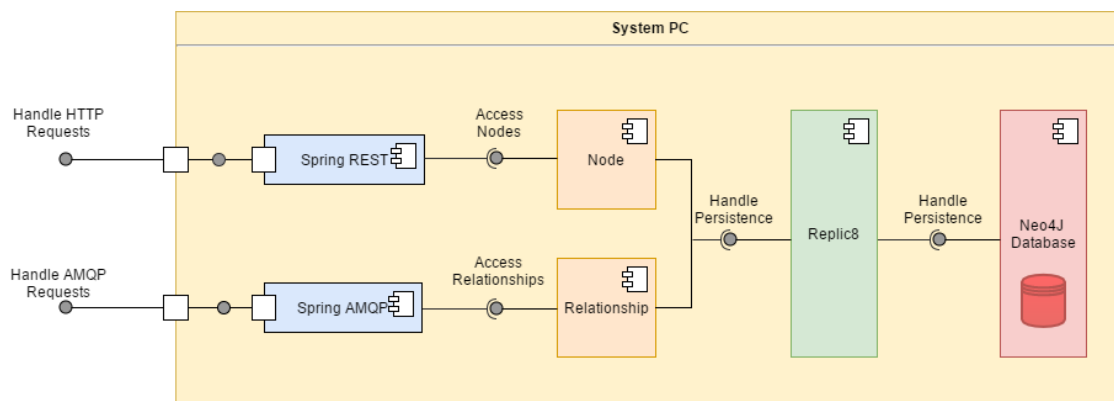


Figure 16 - Replic8 as a Neo4J wrapper UML component diagram

Although possible, and advantageous in some regards, that approach would impose some heavy drawbacks. An advantage would certainly be the transparency for application developers applying the framework, also, this approach would move processing time on slave instances to the persistence layer. On the drawbacks side, Replic8 would become highly coupled to a single database implementation, enforcing constant maintenance and update for each new version of Neo4J in order to maintain compatibility.

These limitations led the move of the transaction interception on Replic8 to a much higher level layer on the application. The idea is to intercept transactions much earlier, in the services/model layer of the application.

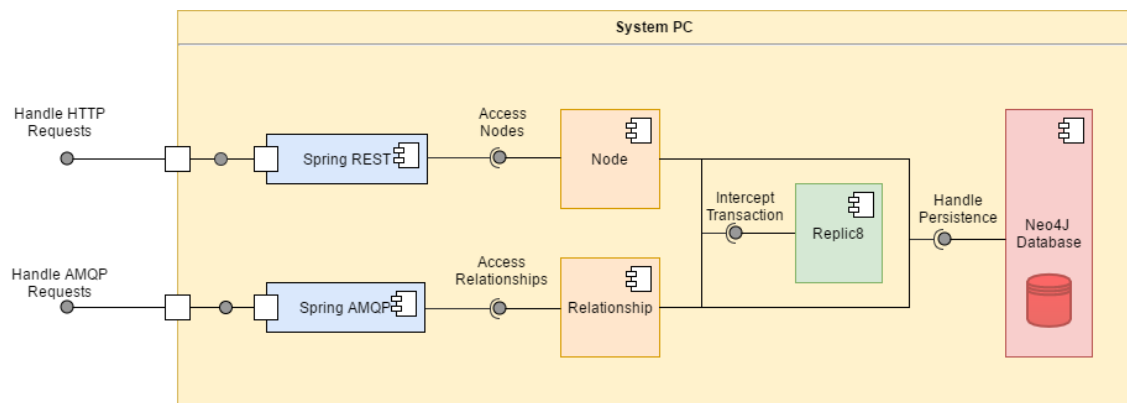


Figure 17 - Replic8 typical positioning in the application UML component diagram

This approach has some drawbacks, but also some big benefits. On the drawbacks side, there is the additional processing effort each instance has to perform, as each instance will process the transaction from an earlier processing point in the application. Another drawback is the loss of transparency for application developers when applying the framework. With Replic8, some slight additions have to be performed in the targeted application when it acts as master. Nevertheless these modifications are not very intrusive, as they basically consist in adding annotations to some service methods and extending an AspectJ abstract class.

On the advantages side, moving transaction interception to a higher layer allows persistence abstraction. Replic8 can be successfully applied to any java application backend regardless of the persistence technology it may use, and not be bound to Neo4J backed applications only. Also, with this approach, is possible to choose only a subset of the domain entities to be replicated, which can be useful to apply replication between heterogeneous systems.



## 5 Development

Early during the development of this project, performing an analysis on the state of the art of replication frameworks and techniques was a major advantage in order to see the big picture before starting the development. Perhaps the more significant technical decision was to go the BASE route. Optimistic replication along with Eventual Consistency was chosen to achieve a compromise between performance and availability. This decision was supported by the fact that the current business needs does not demand 100% synchronized data every time across instances. As such, instead of incurring in all drawbacks and performance penalties imposed by a Pessimistic Replication approach, the natural choice was to go with the more relaxed Optimistic Replication.

In the next paragraphs it will be overviewed some key aspects of the Replic8 framework. Some important technical details and operation flows will be overviewed, along with its usage of some best practices in software development.

### 5.1 Design Patterns

Design patterns, in software development, are ways to structure and arrange objects to solve common design problems (Gamma, 1995). During Replic8 specification/development several design patterns were identified and used.

#### 5.1.1 Creational Design Patterns

The Factory pattern was heavily used in Replic8. Nearly every object instantiation in Replic8 framework is handled by a factory. What a factory basically does is to encapsulate object instantiation and abstract it from the client code.

Complex object instantiation either inside or outside Factories are handled with the aid of Builders, following the Builder pattern. Builders are inner classes that aid in object instance creation using chained *'with...(Object someObject)'* methods.

Singleton pattern was used in some limited specific cases. Although this pattern makes it harder to test, sometimes there is the need to use it. A singleton, is a class that is only instantiated once during the lifecycle of an application. In Replic8, singletons are used to hold application runtime properties and are Enum based, which avoids the expensive usage of a synchronized *'getInstance()'* method usually implemented to protect multiple instantiation from distinct threads.

#### 5.1.2 Structural Design Patterns

The Decorator pattern is usually used to add functionality to an object without changing it. Decorator classes extend the class they are decorating, adding behavior to them. In Replic8

this pattern was used to decorate the base framework *ValidationResponse* class used to validate the entry object in the several communication entry points (Receivers) of the application. Currently there is an *ErlangValidationResponse* decorator class that adapts the response to the structure expected by the specific Erlang implementation of Replic8 senders and receivers.

### 5.1.3 Behavioral Design Patterns

Strategy pattern defines that an algorithm behavior can be selected dynamically at Runtime. Replic8 uses this pattern when persisting transaction recovery logger entries. The *OutgoingTransactionProcessor* class is composed by the *TransactionRecoveryLogger*, an interface. The concrete implementation is not known by *OutgoingTransactionProcessor*, and depending on the configuration it can be a '*FileTransactionRecoveryLogger*' or a '*DatabaseTransactionRecoveryLogger*'. Therefore, depending on the implementation set at Runtime, which is configured in a properties file (detailed in Replic8 Properties chapter), when *OutgoingTransactionProcessor* calls the '*transactionRecoveryLogger.logTransaction(...)*' method, the transaction is either logged in a file, or in a database.

The observer pattern is used by Replic8 when slave instances register into the cluster through the master instance. The subject of observation is the Transaction Broadcaster. The Transaction Broadcaster keeps a set of interested observers (the ones that have registered), until the slave instance goes down. Then, for each transaction to broadcast, the Transaction Broadcaster iterates and notifies all observers with the transaction context.

## 5.2 Unit and Integration Testing

Unit and Integration testing plays an important role in software development. Not only they assure the correct functionality of the currently implemented features, but also prevents functionality/behavior break during further developments. As such, an early goal was set to achieve high test coverage in Replic8.

Replic8 source code is covered by both unit tests, that cover almost all the source code of the framework, but also by some integration tests, in order to test some key functionalities flows.

Currently, Replic8 has about 81% Line Coverage and 65% Branch Coverage. Line Coverage is the overall coverage of all lines of code on the application, while Branch Coverage focus on divergent code paths like if-else or switch statements.

Those measurements were obtained with the Cobertura Maven plugin.

## 5.3 Replic8 Tech Stack

As stated in Requirement C, the framework should be developed in Java programming language. As such its design and development was largely based on Java technology.



Figure 18 - Replic8 Technology stack

Replic8 makes extensive use of Java 8 features like lambda expressions or Streams, taking advantage of its lazy loading nature when handling collections.

AspectJ is a technology used in Java to handle cross cutting concerns. One of the trivial examples of AspectJ use is on application logging. With AspectJ is easy to apply simple context logging to all classes. AspectJ uses a concept called weaving, which adds the functionality declared in an AspectJ class to the target classes, creating a modified version of those classes embedding the AspectJ code. Replic8 uses AspectJ to implement its Transaction Interceptor.

Replic8 uses Erlang to handle communications between instances. Erlang is a language used in telecom industry, created by Ericsson. It was designed to handle heavy concurrency, and uses the actor model for passing messages between Erlang processes.

There was a strong emphasis on unit testing in the framework. Almost every source code class has a correspondent unit test class. As stated, unit test code coverage is 81% for line and 65% for branch (conditionals). For unit testing the framework relies on JUnit to support basic test assertions, on Mockito to Mock simple plain old java object (Pojo) classes, and PowerMock to mock Singletons, enums, protected and private methods.

### 5.3.1 Why Erlang for communication?

Remote communications in Java is a mature theme and there is no shortage of options. Java ecosystem has all kinds of libraries and plugins to handle remote communications, ranging from Simple Object Access Protocol (SOAP) to Java Remote Method Invocation (RMI), passing through integrations with message brokers like ActiveMQ, RabbitMQ (actually written in Erlang) or Java own Java Message Service (JMS) which is backed by the likes of IBM WebSphere or Oracle Weblogic application servers for the broker infrastructure.

With all these options, why was Erlang chosen? Erlang may not be the best language for math calculations, or heady data processing like images and video processing, but among other



things it excels on concurrency, as its uses pure message passing between processes resulting in absolutely no shared memory between them. Also, message ordering is ensured by Erlang Runtime as long as it is delivered to the destination process. These two properties alone are a big plus for Replic8 framework, as the master instance will be constantly broadcasting messages to slave instances.

Another essential feature provided by Erlang, that is extremely useful for Replic8 framework is process linking. With process linking two processes can be linked, and if one of them dies, the other receives a notification. This is especially important to the Health module, where health check processes for each slave instance are linked to the one on the master. If one end of the link goes down, the other is notified.

To summarize, Erlang was chosen to handle communications because of its powerful concurrent nature and its asynchronous message passing between processes. Providing message ordering and process linkage is also a major plus.

## 5.4 Replic8 Modules Breakdown

Replic8 four modules are composed of several components. In the next pages those components will be further detailed. The following image provides a high level overview of those components, and their relation with both roles (master/slave) that Replic8 can assume on a transaction flow.

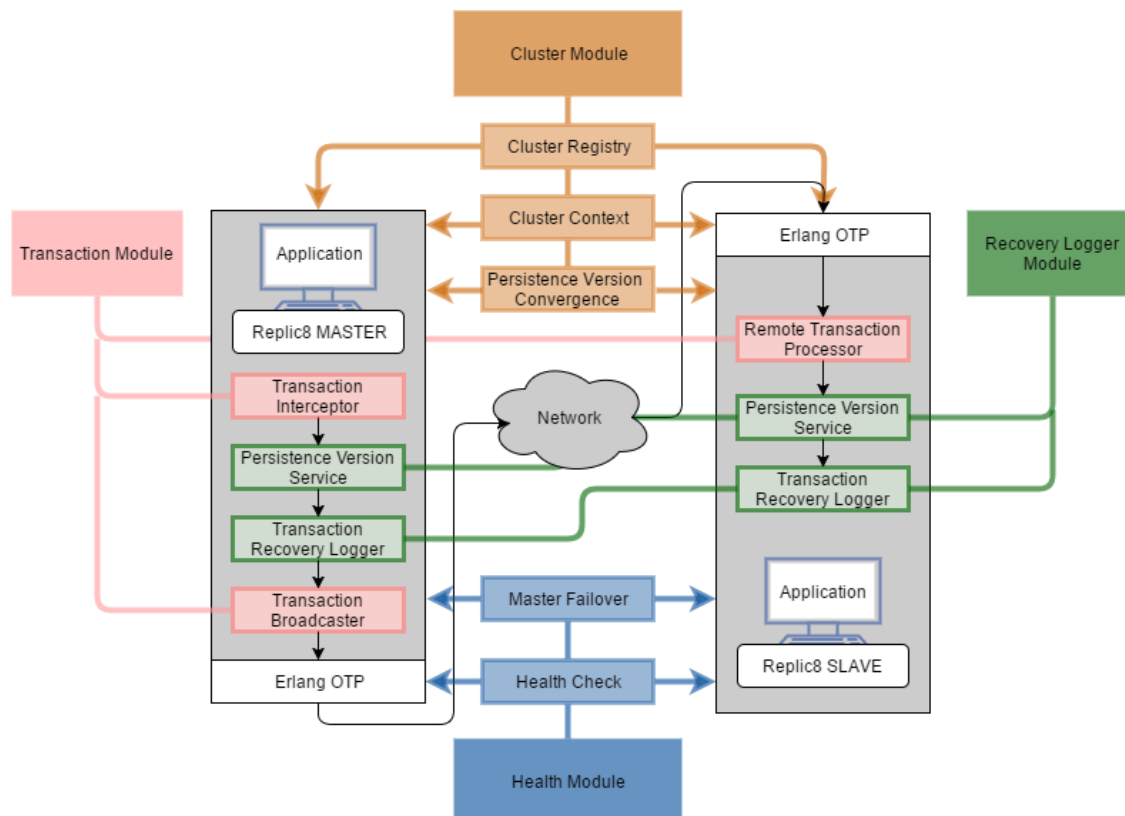


Figure 19 - Replic8 high level component overview

### 5.4.1 The Transaction Interceptor

Intercepting transactions is the most important single feature on any replication framework. Before broadcast to other instances, the framework has to be able to capture the transaction context, and the transacted data. Unlike other frameworks, Replic8 captures the transaction at the middleware level instead of database driver level. This solution solves the problem of capturing transactions on an embedded database, at the same time that provides much more flexibility as the framework is not dependent on the database technology.

Replic8 usually is set to intercept the transaction at the service layer of an application. In a Model View Controller (MVC) application, the service layer usually sits between the controller and the models. The service layer is often used to orchestrate calls between one or more models within a transaction.

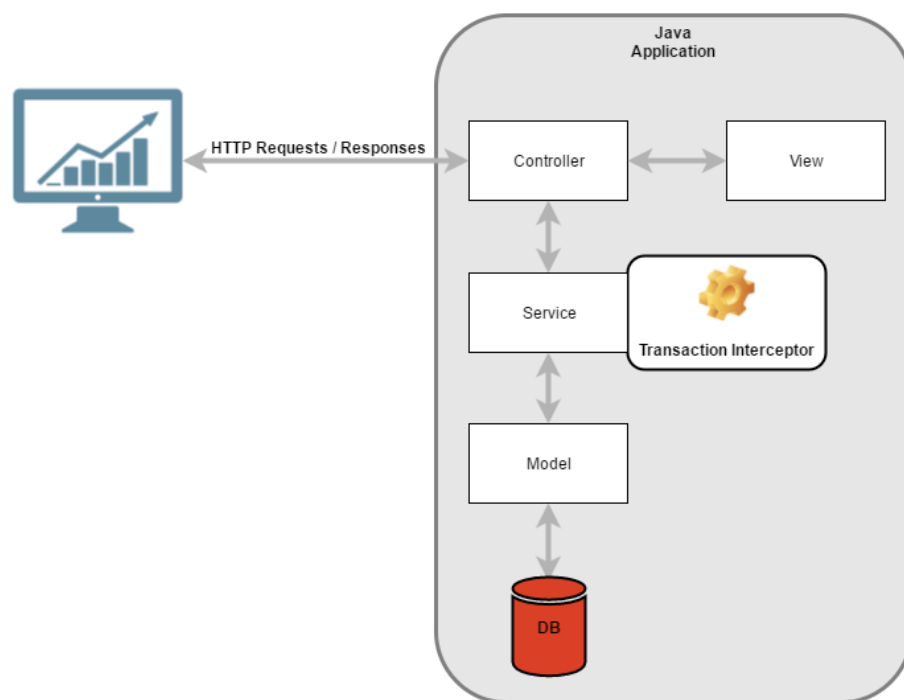


Figure 20 - Transaction Interception with Replic8

Replic8 framework uses Aspect Oriented Programming (AOP) a programming paradigm for handling cross cutting concerns as logging and authorizations. The term AOP was coined in Xerox Palo Alto Research Center in 1996 and one of the first implementations to see the light of the day was AspectJ, an implementation of AOP for the Java programming language (Laddad, 2003).

What AspectJ does is add functionality on top of already existing source code classes, without having to modify them. An Aspect can be seen as some kind of decorator that executes some additional logic around already existing logic in some class, or group of classes. AspectJ uses the AspectJ Compiler (AJC), a superset of the Java compiler to perform an action called "weaving". Weaving is the process of merging both the source code and the logic contained in an Aspect class into one new class.

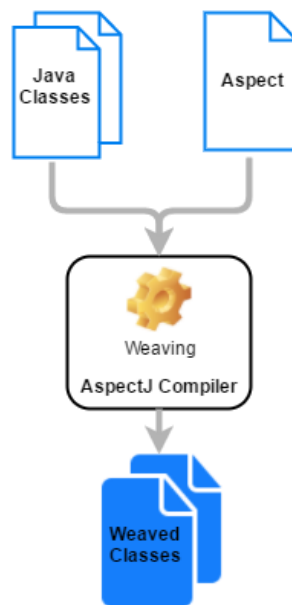


Figure 21 - AspectJ weaving overview

Replic8 provides its Transaction Interceptor in the form of an abstract AspectJ class. To make use of Replic8, client applications must extend this class in order to define which classes will be targeted by Replic8 for transaction interception.

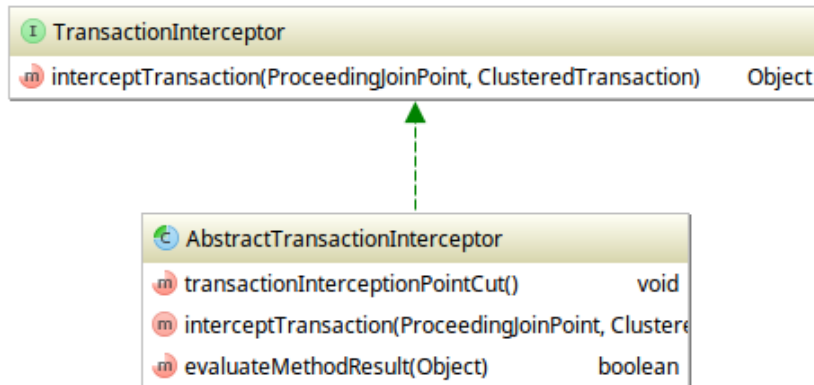


Figure 22 - Transaction Interceptor UML class diagram

The *AbstractTransactionInterceptor* class is the core class of the framework. It is composed by two abstract methods that client applications should implement. The first, '*transactionInterceptionPointCut()*' represents an AspectJ pointcut definition. A pointcut definition is basically an empty method that is annotated with the pattern needed to match the classes and methods to be weaved by AspectJ. The targeted classes for this pointcut, should be the ones at the service level layer or equivalent. Methods that should be handled by the Transaction Interceptor should be annotated with the *@ClusteredTransaction* Java annotation provided by Replic8 framework.

The second method '*boolean evaluateMethodResult(Object)*', should be implemented by client applications, containing the verification logic for the method result. It is expected that

the returning objects for the targeted classes in *'transactionInterceptionPointCut()'*, unequivocally represent an transaction failure or success.

The *AbstractTransactionInterceptor* contains the remaining logic to act accordingly with the outcome of the *'boolean evaluateMethodResult(Object)'*. If the evaluation succeeds, the transaction is propagated to the slave instances. If the transaction is not succeeded, the process exits.

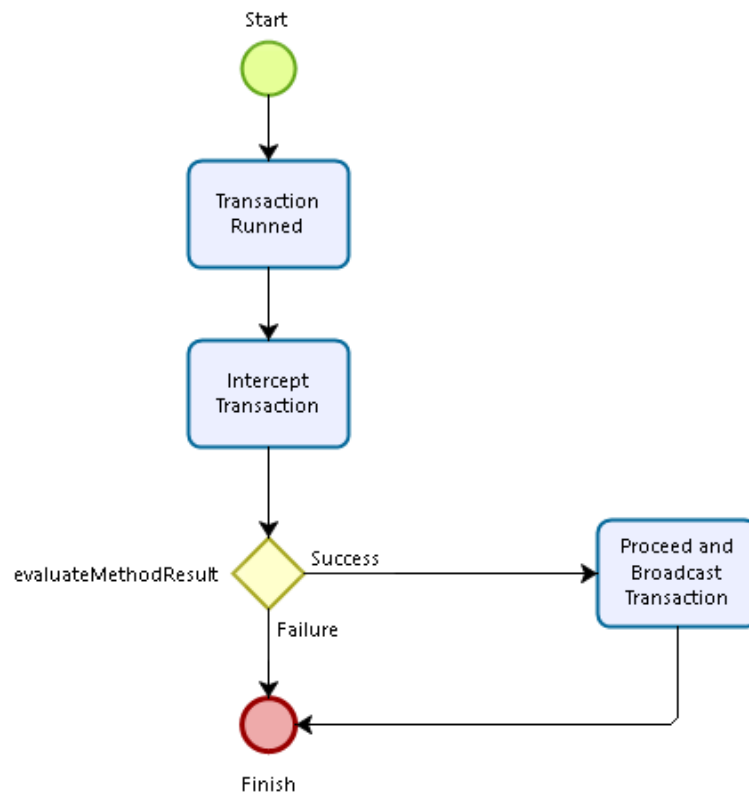


Figure 23 - Transaction Interceptor flow UML activity diagram

The full class diagram for the main entities involved in broadcast the transaction to the remote observers can be seen in the next figure.



Figure 24 - Transaction Interception involved classes UML class diagram

## 5.4.2 Transaction Broadcaster

The Transaction Broadcaster is the component responsible to broadcast transactions intercepted, and evaluated as committed by the Transaction Interceptor. This component employs the observer pattern and is responsible to maintain a set of interested observers. Each observer represents a Replic8 slave instance that has been previously registered in the cluster. When a slave instance becomes unavailable, the observer is unregistered.

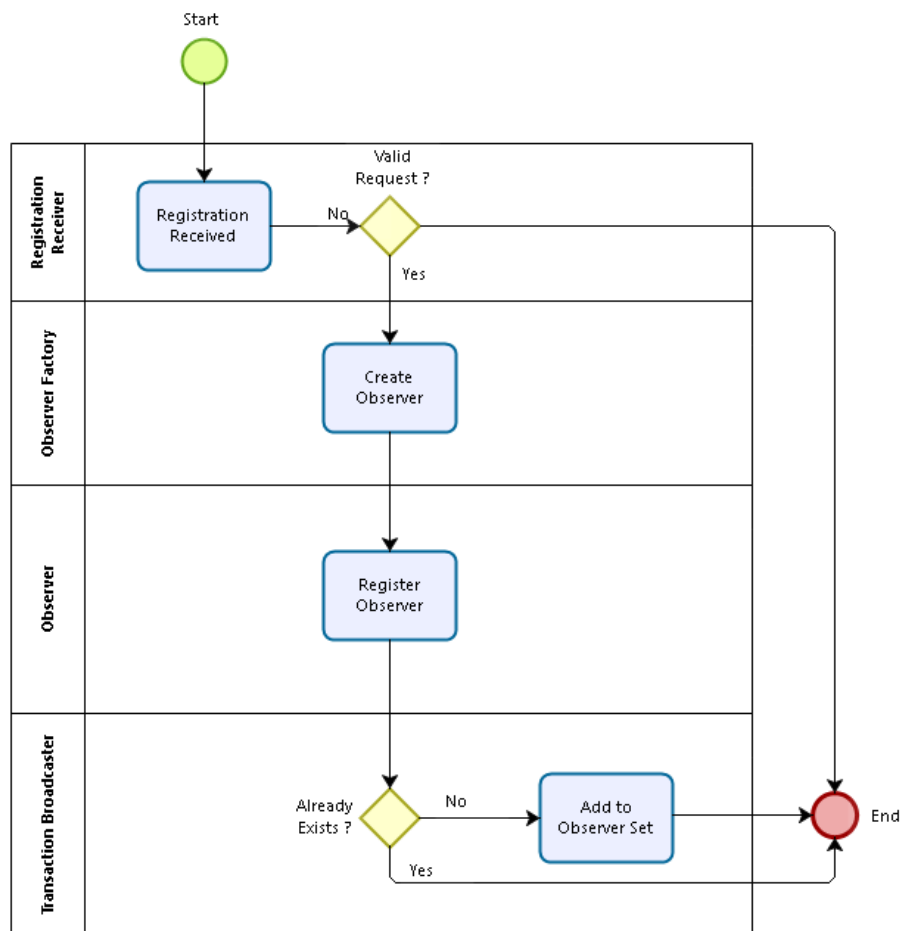


Figure 25 - Observer Registration Flow UML activity diagram

After receiving a valid registration request, an Observer Factory creates the observer using the registration request parameters and passing the Transaction Broadcaster instance reference. Upon instantiation, the Observer registers itself on the Transaction Broadcaster. The Transaction Broadcaster then adds the Observer to the collection of registered Observers. A Java Set is used to maintain the list of Observers in the Transaction Broadcaster meaning that no duplicate observers will be registered.

The following class diagram details the implementation of the observer pattern on Replic8 for the Transaction Broadcaster (the subject) and the Transaction Observers.

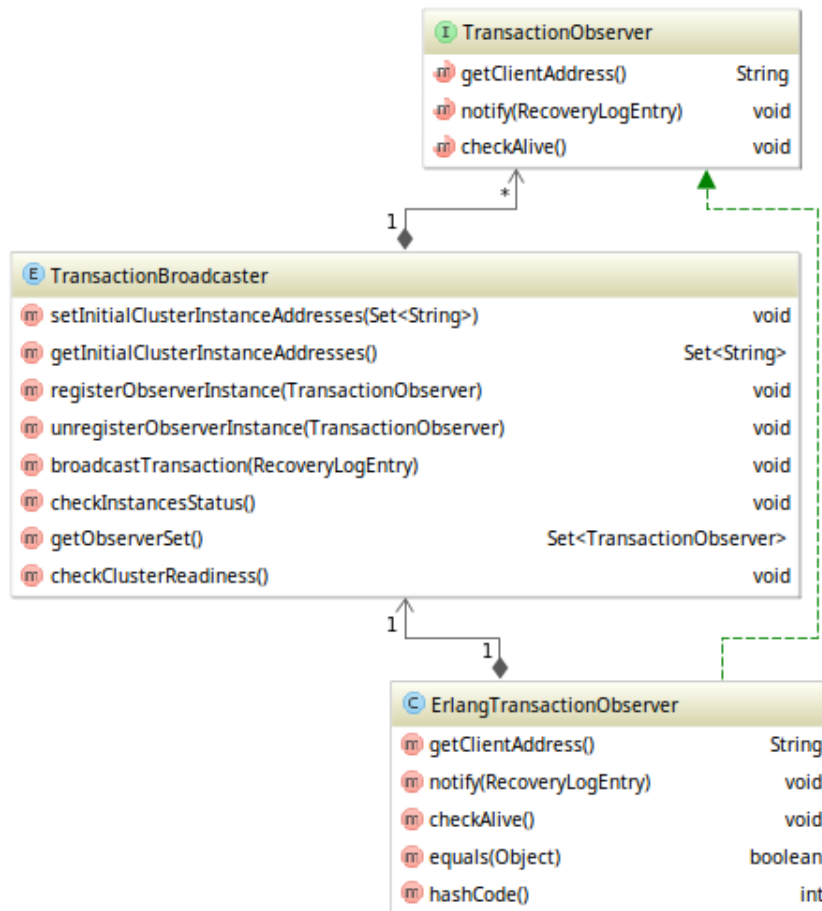


Figure 26 - Transaction Broadcaster UML class diagram

From the above class diagram, it can be seen that The Transaction Broadcaster ‘has a’ set of Transaction Observers. *ErlangTransactionObserver* class is a concrete implementation of the *TransactionObserver* Interface and is composed by one *TransactionBroadcaster*.

When a transaction arrives at the Transaction Broadcaster, the observer set is iterated and the transaction context is sent to each one of the registered observers. Each observer instantiates a new Transaction sender that will be responsible to handle the communication between the master instance, and the correspondent slave.

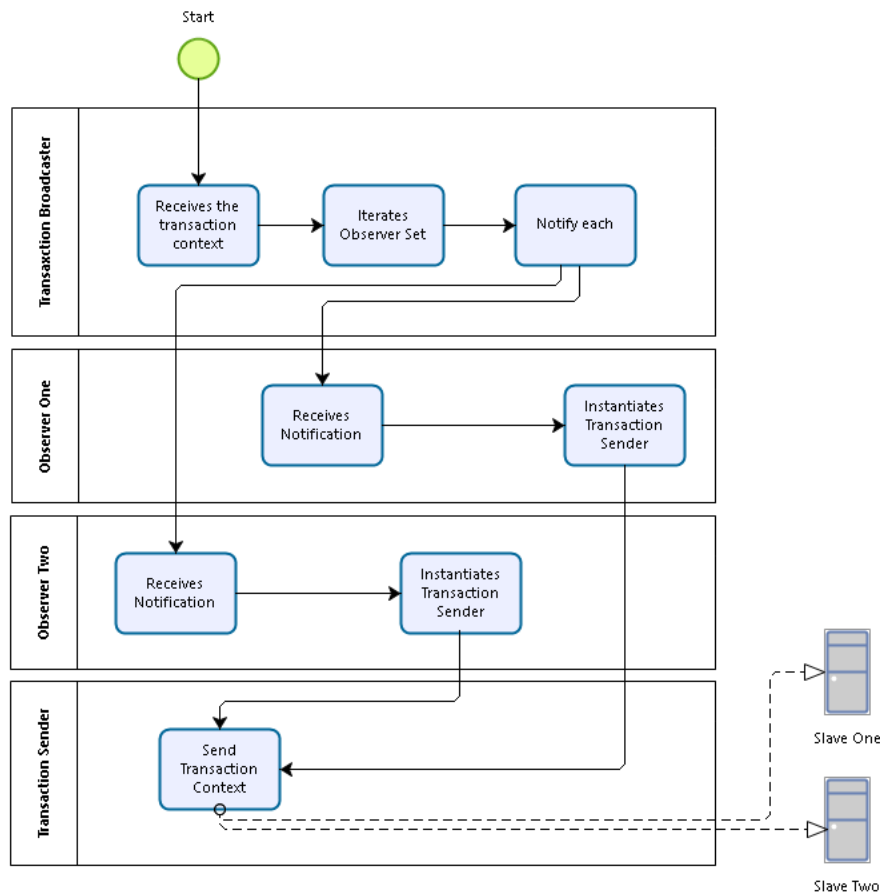


Figure 27 - Transaction Broadcast flow UML activity diagram

### 5.4.3 Remote Transaction Processor

Slave instances receive transaction contexts through Transaction Receivers. These are specific classes for receiving communications from the master instance about a broadcasted transaction. After validating the received transaction context, the receiver invokes the Remote Transaction Processor, who is responsible to verify if the transaction is suitable for processing on the local slave instance.

The transaction context received by slave instances are, in fact, recovery log entries generated at the master instance. When the master processes a new transaction, it also persists a new entry on its local recovery logger along with the correspondent persistence version. This entry is represented by a *RecoveryLogEntry* object that contains all the information that slave instances need to apply the transaction locally.

Before processing the transaction, the RTP first verifies if the persistence version that comes with the received *RecoveryLogEntry*, matches the next to be generated local persistence version. If the persistence versions do not match, the RTP instructs the request of newer transactions than the current local persistence version to the master. If the persistence versions match, then the RTP applies the transaction locally, and commands an update to the local transaction recovery logger. These two last operations are within a local transaction scope.



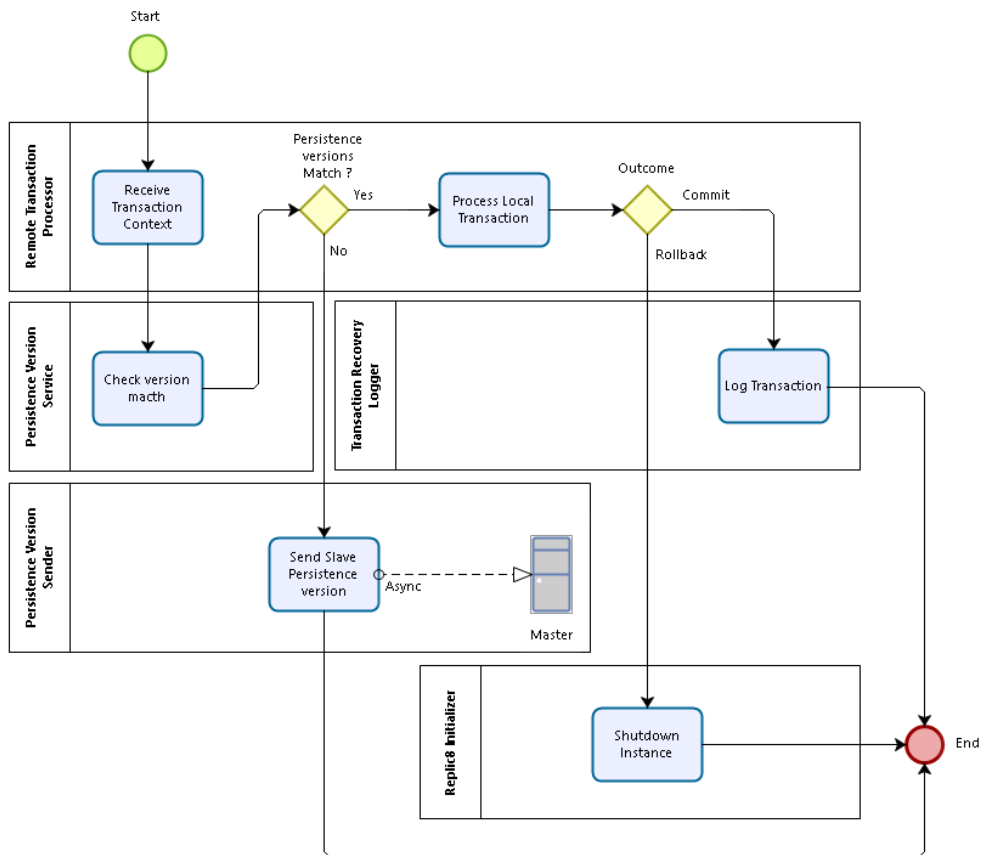


Figure 28 - Remote Transaction Processor flow UML activity diagram

The RTP component uses reflection to apply the transactions locally. Within each *RecoveryLogEntry*, comes information about the service class, the method and the entry object that originated the transaction on the master instance. RTP then uses those three values to reflectively invoke exactly the same call on the slave side.

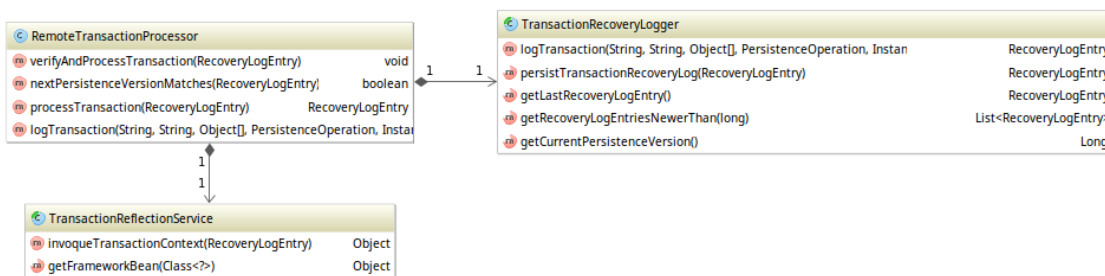


Figure 29 - Remote Transaction Processor UML class diagram

RTP makes use of a *TransactionReflectionService* class to reflectively apply the transaction locally. This abstract class exposes an abstract method called '*getFrameworkBean(Class class)*' which is meant to be implemented by client applications that use dependency injection frameworks like JavaEE (Enterprise Java Beans) or Spring Framework (Spring Beans) in order to retrieve an already instantiated bean within the framework lifecycle.

#### 5.4.4 Persistence Version Service

The Persistence Version Service is actively used by both the master and slave instances. This is a small component that is responsible to handle persistence version related operations. Its main responsibility is to generate the next local persistence version, and also, if in master mode, to handle persistence version convergence requests from slaves. To achieve persistence version convergence, the PVS reads all the local transactions in the master with a persistence version higher than the slave one, and orders their broadcast through the Transaction Sender.

The next diagram explains how persistence version convergence between a slave and the master instance is handled.

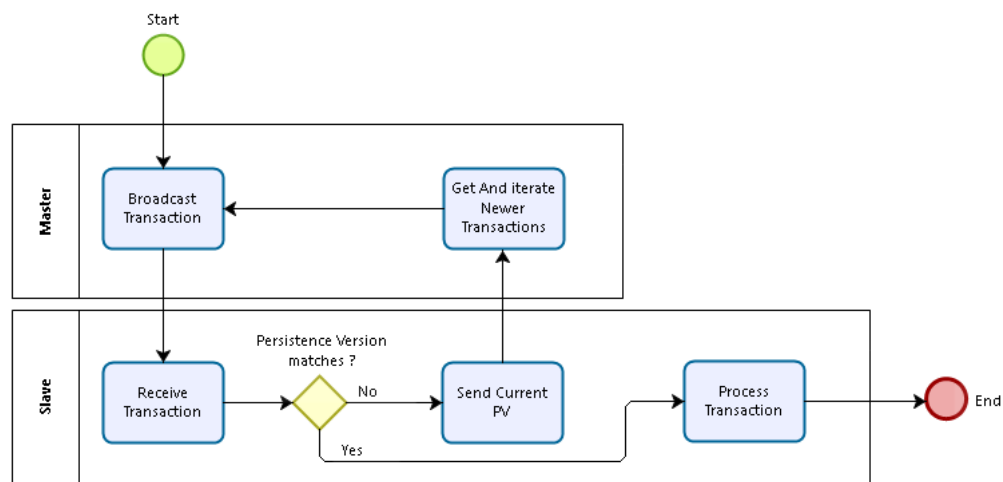


Figure 30 - Persistence version convergence flow UML activity diagram

As a simple example, if the master instance broadcasts transaction 55 to a slave who's the last local transaction persistence version is 53, then the slave will send persistence version 53 to the master. The master will then broadcast transactions with versions 54 and 55 (again) to the slave. If the slave in the meantime receives a transaction with version 56, it requests the same version (53) to master. When it receives the transactions with versions 54 and 55 from the first request, the slave processes them, and only processes the transaction version 56 when it receives the three from the last request (versions 54, 55 and 56), as the first two were already processed.

#### 5.4.5 Transaction Recovery Logger

The Transaction Recovery Logger is the component responsible to maintain the history of the locally persisted transactions. This component plays a crucial role both on master and slave instances.

As stated, TRL is responsible to persist an ordered history of the locally processed transactions. It also assists the Persistence Version Service, providing methods for retrieving the current persistence version and the last logged transaction. It also helps to return the logged transactions newer than a specific persistence version. This last method is used by the PVS

when in master mode to broadcast newer transaction to slaves that reported being on an older version.

The TRL persists transaction history in each instance in a csv file format. However it is designed to easily be implemented for other types of persistence as the classes that use the TRL do not know its implementation until Runtime.

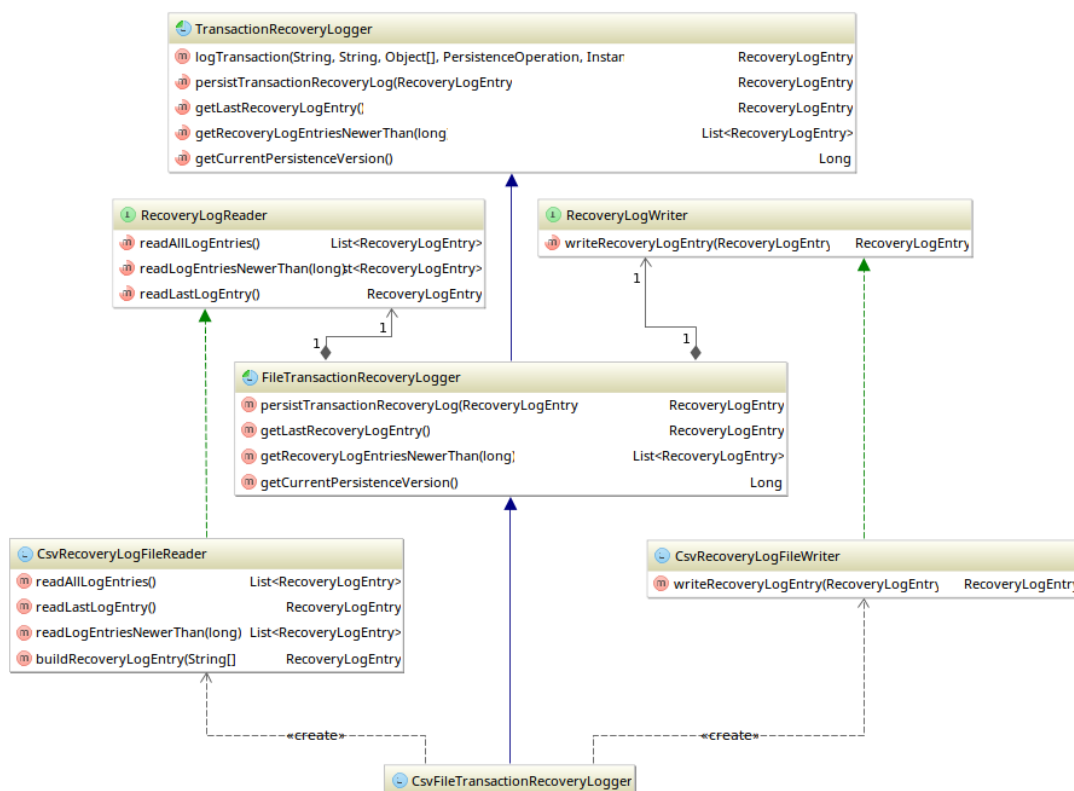


Figure 31 - Transaction Recovery Logger UML class diagram

Components that use the TRL obtain an instance using the *TransactionRecoveryLoggerFactory* class. This class instantiates the type of *TransactionRecoveryLogger* (CSV or other) depending on the configuration present on Replic8 properties.

The recovery log itself is composed of several log entries that map to an object called *RecoveryLogEntry*. Each one of those entries contains the local time at which the transaction occurred, the names of the class and method of the service class where the transaction first began, the serialized object data sent to the service class, the operation type (Create, Update, Delete), and finally, the correspondent persistence version.

When the master broadcasts a transaction to slave instances, in reality it sends a serialized representation of the generated *RecoveryLogEntry* for that same transaction.

## 5.4.6 Cluster Registry

Cluster registry handles the process of registering slave instances within the cluster. Slave instances can be configured either in passive or active modes. When in active mode, the slave

instance is configured with the address of the cluster's master instance. In this case as soon as the slave application is up, Replic8 sends a registration request to the master.

Slaves can also be configured in passive mode. This mode is used when the master is not yet initialized and therefore no master address is configured in the slave. As soon as the master starts, it connects to each configured slave, and sends it the cluster context. Passive slaves will then send a registration request to the master instance that is referenced in the cluster context they received.

#### 5.4.6.1 Replic8 Initialization as Master

When the application instance starts, Replic8 will also start with the configured role. If the instance role is master, then Replic8 upon startup loads a configuration xml file that contains information about itself and about the slave instances that should be part of the Cluster. Information for each instance includes the role, the instance network address and a description.

After loading instance configurations, Replic8 will iterate through all configured slave instances, and send to each of them information about itself (master instance) and the other slave instances in the cluster. By enabling slave instances to be aware about other players in the cluster, they will have all the information required in the case they need to assume the master role in the advent of a master failure.

After sending the cluster composition information to slave instances, the master itself waits for every slave instance to register in the cluster. When a transaction is committed on master, every registered slave receives the correspondent *RecoveryLogEntry* for the transaction context. Slaves that did not yet registered will receive transactions from the point they register within the cluster. It is expected that the late joining slaves will be out of sync in terms of persistence version, and as such, slaves in such situations will eventually converge as seen in [Remote Transaction Processor](#) chapter.

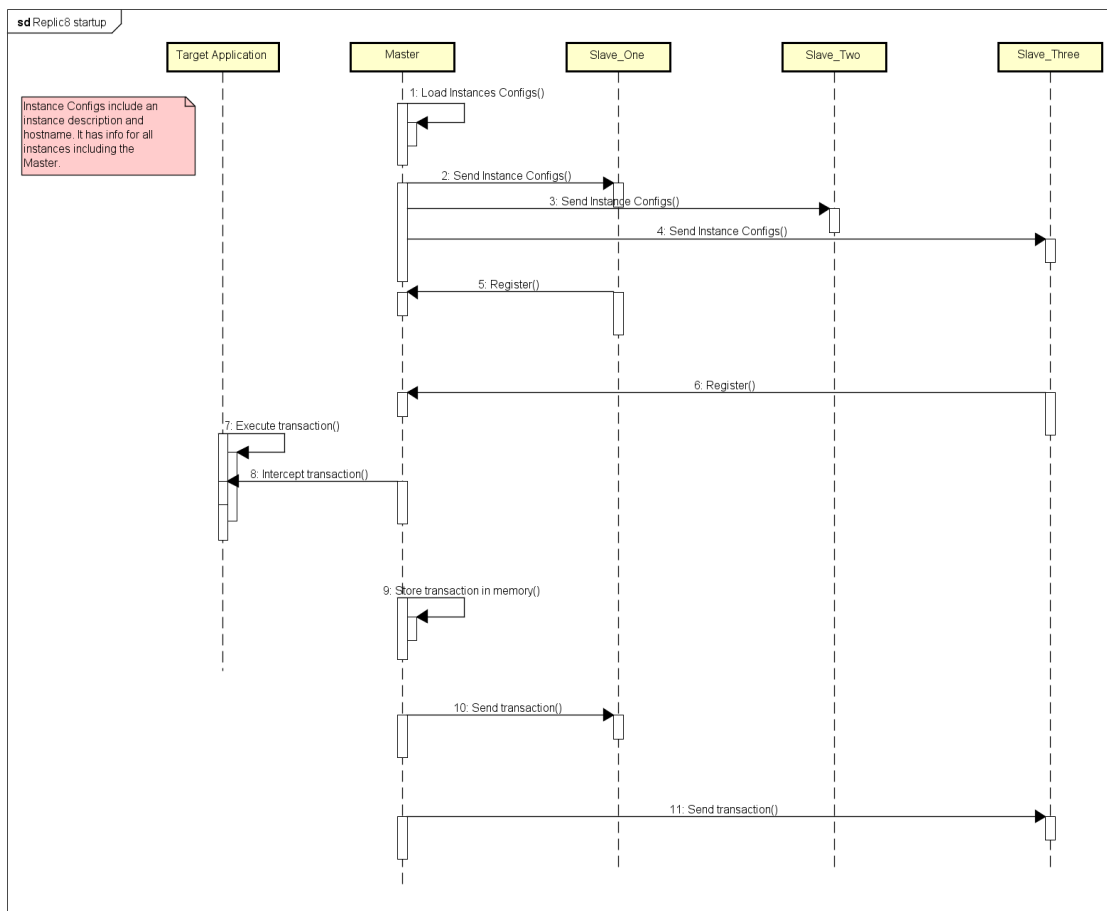


Figure 32 - Replic8 initialization as Master UML sequence diagram

#### 5.4.6.2 Replic8 Initialization as Slave

Replic8 can be initialized as slave with two different options, either in passive or active mode. When in passive mode, slave instances will always wait for the master to send the cluster configurations before they register in the cluster. Passive slaves can take the master role as they know the initial cluster composition; however they need to be started before the master. When in active mode, the slave will be the one to contact the master first to request registration. Active slaves can be started any time after the master, however they cannot promote themselves to master because they do not know the entire cluster composition. Also, if the master fails and one of the passive slaves takes the role, active instances will not receive transactions until restarted. This is a limitation in the current version of the framework that will be addressed in future versions.

#### 5.4.6.3 Slave Registration Request

When a slave instance issues a registration request to master, it also sends the local persistence version (LPV). This version represents the last transaction that the instance has record of. As such, when receiving a registration request from a slave instance, the master will check its own LPV and compare it to the one sent by the slave. If the LPV of the slave is lower

than the master sends to the slave instance all the newer transactions that occurred since the slave LPV.

### 5.4.7 Cluster Context

The cluster context represents the information about every single instance that is part of the Replic8 cluster. This information includes descriptions, roles and network addresses for all instances, including the master.

Upon initialization the master instance broadcasts this information to all the passive slaves that were already started. Providing this information to the slaves is essential to perform a master failover in case of a failure of the master instance.

The information regarding the cluster context is configured in the master in the form of a XML configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterInstances>
  <!-- Master instance -->
  <clusterInstanceDefinition role="MASTER">
    <description>Master Instance</description>
    <address>master@localhost</address>
    <failoverHierarchy>1</failoverHierarchy>
  </clusterInstanceDefinition>
  <!-- Slave instances -->
  <clusterInstanceDefinition role="SLAVE">
    <description>Slave One Instance</description>
    <address>slave_one@localhost</address>
    <failoverHierarchy>2</failoverHierarchy>
  </clusterInstanceDefinition>
  <clusterInstanceDefinition role="SLAVE">
    <description>Slave Two Instance</description>
    <address>slave_two@localhost</address>
    <failoverHierarchy>3</failoverHierarchy>
  </clusterInstanceDefinition>
</clusterInstances>
```

Figure 33 - Cluster context configuration file example

In the configuration file, an entry is added for each instance that composes the cluster. Each instance entry identifies the instance role, and contains information about its description, the remote address and the position it occupies in the cluster hierarchy, which is used for master failover.

### 5.4.8 Master Failover

Master Failover is the process of instance replacement when the master instance goes offline. As stated in [Health Check](#) chapter, the next slave instance in the hierarchy (master candidate) sends periodically health checks to the master instance.

When a series of health checks result in a timeout, the slave instance assumes control of the cluster, and re-sends the transaction context to all the remaining slave instances. The next in succession instance also changes to the slave instance that has the closest higher 'failoverHierarchy' value.

If the master instance, suddenly fails after committing a local transaction and before broadcasting it to the remainder cluster, all the other instances, including the new master will be one transaction behind the original master. If the original master becomes online again, it will be out of sync with cluster. As there is no way for Replic8 to know if a failing master broadcasted all its transactions before failing, its database should be restored to the initial state before it can join the cluster again. By initial state, it's either a fresh empty database, if Replic8 was setup on a new system, or a database dump from the time were Replic8 was integrated in the system. In this specific scenario, the transaction that was not broadcasted by the original master is lost.

#### **5.4.9 Health Check**

Replic8 periodically send health checks to each instance of the Cluster. Master sends health checks for each slave, and the slave that is the next master in the hierarchy sends health checks to the master. Other regular slaves, can also send health checks to master, when they receive a transaction with a persistence version that do not match their next local persistence version.

Health checks coming from the master instance do not contain any data. When a slave receives a health check, it replies with an OK and with its current local persistence version.

Health checks coming from the slave, contains the local persistence version of the Slave. When the master receives the health check from a slave, it checks the slave's local persistence version and executes the Persistence Version Convergence process as stated in both [Persistence Version Service](#) and [Persistence Version Convergence](#) chapters.

#### **5.4.10 Persistence Version Convergence**

Persistence Version Convergence is performed at several stages. When a slave requests a registration in the cluster, when a slave receives a new transaction, and when a health check is sent from the slave to master.

PVC in Replic8 is handled as an iterative process, convergence can be achieved with one simple iteration or several. Taking the example from [Persistence Version Service](#) chapter, the next diagram provides a sequence overview of the aforementioned example.

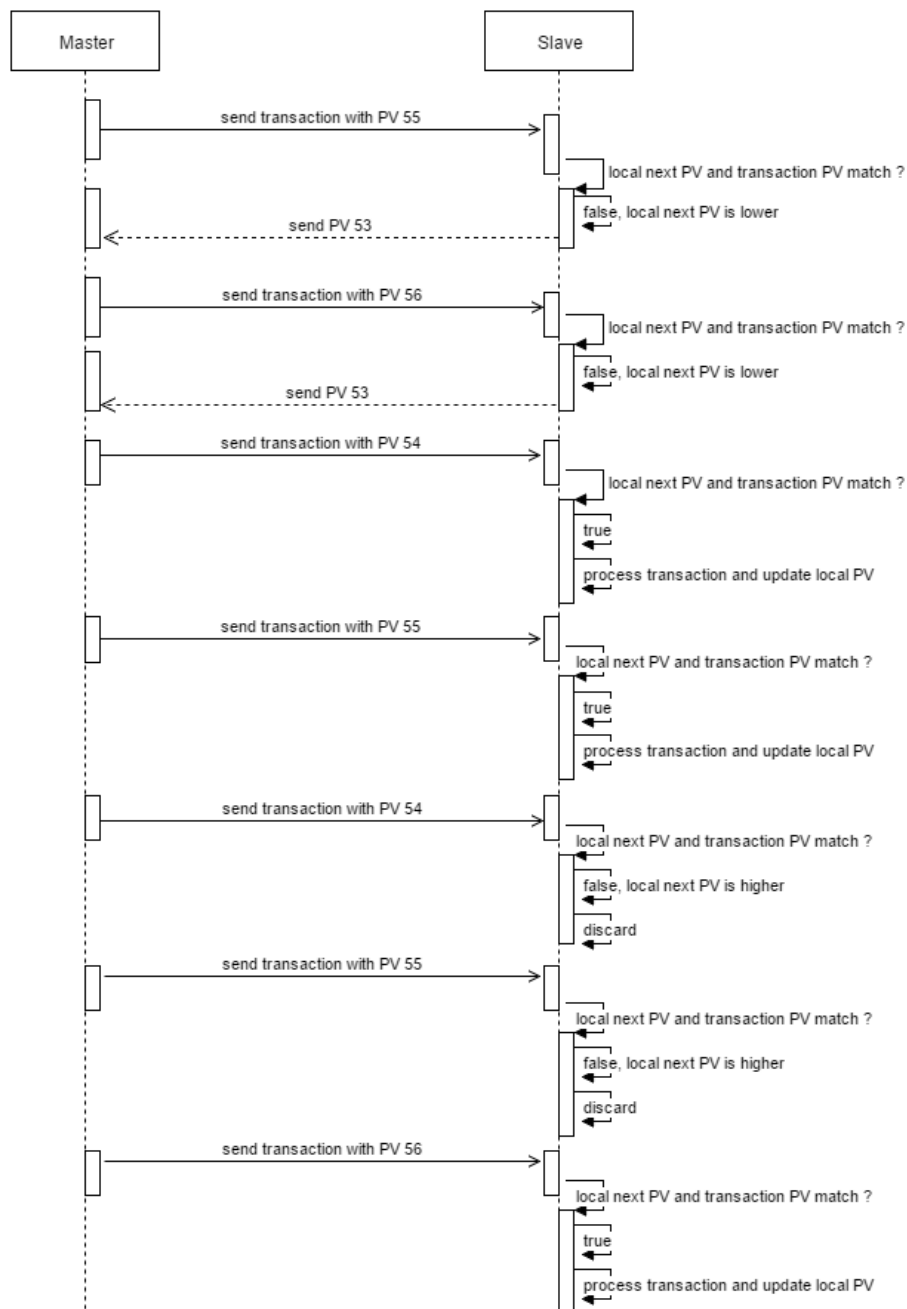


Figure 34 - Persistence version convergence UML sequence diagram

#### 5.4.11 Senders and Receivers

Replic8 is composed by a series of sender and receivers in order to coordinate all cluster operations. For example, transactions are broadcasted by the master instance using a transaction sender and received by the slaves with a transaction receiver. In Replic8 there is a pair of sender/receive for each transmitted message type within the cluster.



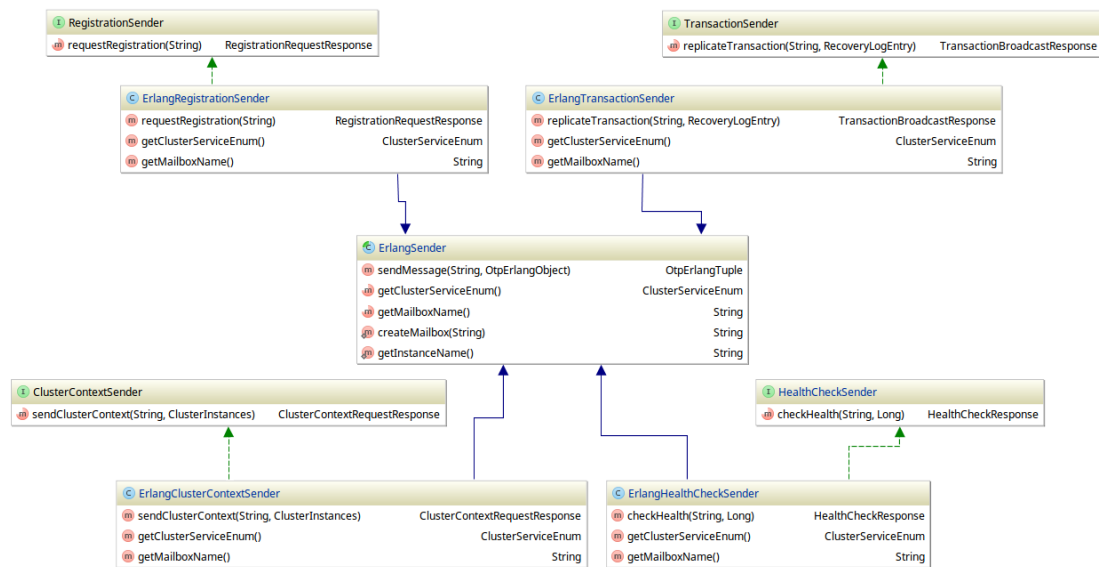


Figure 35 - Replic8 senders UML class diagram

Each sender implements a specific interface that defines the basic sender contract. As stated before, the default remoting technology adopted for Replic8 is Erlang OTP, as such Replic8 provides the correspondent Erlang concrete senders for each sender interface.

In this case, all the concrete implementations extend the abstract class *ErlangSender* which aggregates all the communication logic shared between all the concrete Erlang senders, only delegating to them, sender specific operations.

As Replic8 follows the object oriented programming design principle ‘program to an interface, not implementations’ Client classes that interact with the senders are not bound to a specific implementation, and do not know until runtime, which one is used.

On the receiver’s side, the pattern is the same as seen with the senders.

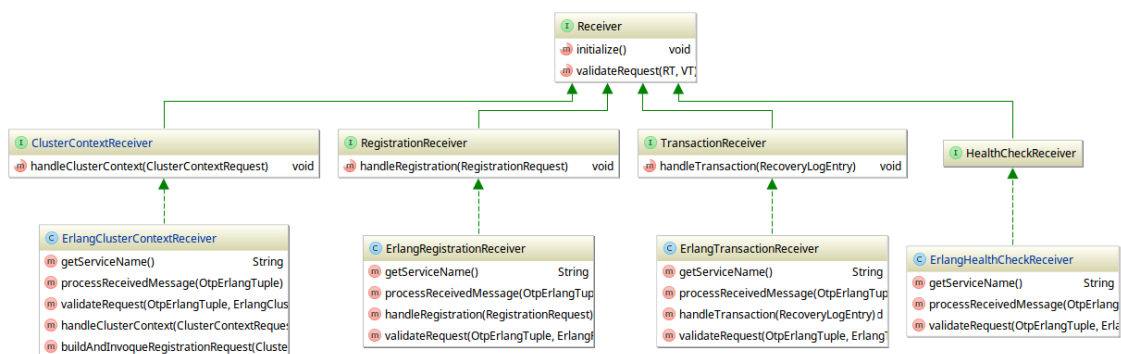


Figure 36 - Replic8 receivers UML class diagram

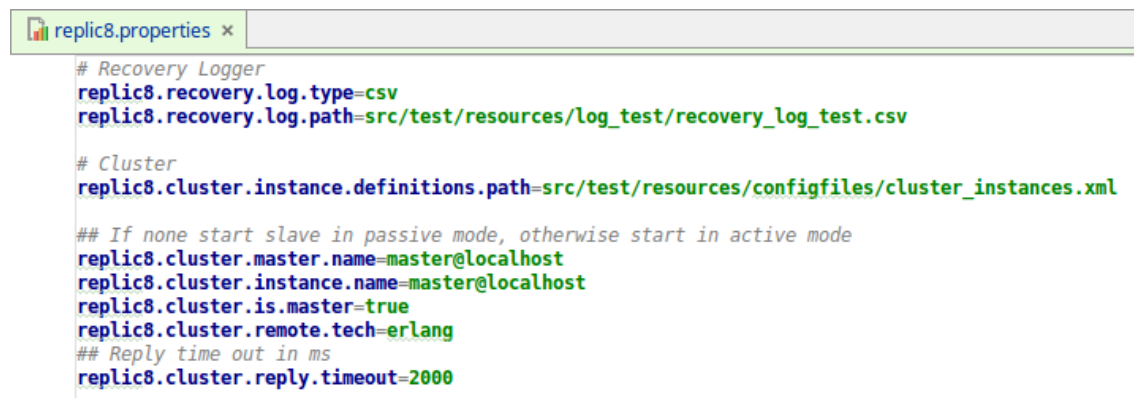
As stated in before, the use of specific senders and receivers depends on which role Replic8 is playing in each instance. The senders *TransactionSender*, *ClusterContextSender* and

*HealthCheckSender* are the only used by Replic8 when performing the master role, while when in slave role only *RegistrationSender* and *HealthCheckSender* are used.

On the receivers side, when in master role Replic8 uses the *RegistrationReceiver* and *HealthCheckReceiver*. When configured to slave role Replic8 uses the *ClusterContextReceiver*, *TransactionReceiver* and *HealthCheckReceiver*.

## 5.5 Replic8 Properties

In order to configure Replic8 either with a master or slave role, a set of properties must be configured. Figure 37 presents the available properties to be configured for a master or slave role.



```
# Recovery Logger
replic8.recovery.log.type=csv
replic8.recovery.log.path=src/test/resources/log_test/recovery_log_test.csv

# Cluster
replic8.cluster.instance.definitions.path=src/test/resources/configfiles/cluster_instances.xml

## If none start slave in passive mode, otherwise start in active mode
replic8.cluster.master.name=master@localhost
replic8.cluster.instance.name=master@localhost
replic8.cluster.is.master=true
replic8.cluster.remote.tech=erlang
## Reply time out in ms
replic8.cluster.reply.timeout=2000
```

Figure 37 - Replic8 properties file

### 5.5.1 Recovery Log properties

The first two properties are related to the Replic8 recovery logger. As previously seen, the recovery log file holds a history of each transaction handled both by master and slave Replic8 instances. The 'replic8.recovery.log.type' property is used to select the type of recovery log file to use. As the current Replic8 prototype only has support for \*.csv file type, 'csv' must be chosen. The property 'replic8.recovery.log.path' is used to instruct where to store the recovery log file.

### 5.5.2 Cluster properties

The following properties are all related to the cluster behavior. The first one, 'replic8.cluster.instance.definitions.path' is used to point out which file Replic8 should look to load the cluster definitions. Only the master instance loads these definitions in order start the cluster for the first time. Nevertheless, as stated earlier in chapter 5.4.7 Cluster Context, slave instances use the specified file to store cluster definitions send by the master, so they can be aware of the remaining cluster in order to assume the master role in case of a master failure.

The second property `'replic8.cluster.master.name'` is only used by slaves when started in active mode (see chapter 5.4.6.2 Replic8 Initialization as Slave), so they can proactively connect to the master instance to join the cluster. This property is not used when Replic8 is started as master or as a passive slave.

The property `'replic8.cluster.instance.name'` is used to define the name of the Replic8 instance. The names configured in this property for all instances are the ones that must be also present in the cluster context configuration file. Property number four, `'replic8.cluster.is.master'` is used to select whether the instance should be started with a master role (true) or with a slave role (false).

The fifth property `'replic8.cluster.remote.tech'` is used to select the type of remote technology to be used between Replic8 instances. Currently, in its current prototype state, Replic8 only supports Erlang, but other remoting technologies can be easily added.

The last property `'replic8.cluster.reply.timeout'` specifies the amount of time each instance sending a message should wait for an acknowledgement from the other side.

## 5.6 Limitations

The proposed solution, as an early prototype version, has some limitations. This chapter exposes and describes some of those limitations.

Being a pure middleware based solution, for each database replica, there must be an application instance configured with Replic8. This is true not only for applications with embedded databases like System PC, but also for applications backed by other type of databases. Although the need for more application instances can rise hosting and maintenance costs, in the specific case of System PC, the Neo4J HA solution would impose the same drawback, while also adding its licensing fee costs.

When using Replic8, information managed by scoped entities, like Session or Request scoped Java Beans on the master instance, are not transmitted to the slave instances. System PC does not maintain such states, and as such is not impacted by this limitation. Nevertheless applications that needs to access these type of beans on service or model classes will not be able to use the current version of Replic8.

As already stated, Replic8 is entirely written in Java, and it uses AspectJ which recompiles source Java code with added functionalities. As such, its usage is limited to Java applications. Although such limitation is not really a disadvantage when comparing with the frameworks studied in the Existing Frameworks sub-chapter of the State of the Art, it narrows the possible applications of Replic8 framework.

Replic8's transaction recovery logger stores each transaction information in the log file, including the serialized representation of the business object. This serialization is handled by Java own *ObjectOutputStream* class that serializes the object representation as a sequence of bytes. For this to work, every business object and all its inner objects should implement the java *Serializable* interface. Although Java serialization allows us some degree of modification in those business objects without breaking the deserialization, special care must be taken when deploying the changes. For example, if new fields are added to the business object, the

slave instances should be updated first. If fields are removed from the business object, then, master instance should be updated first.

Currently, as already stated, active slave instances do not receive the cluster context, and therefore cannot assume the master role. This limitation was based on an early premise that an active slave would be configured in some limited situations, to easily replace a faulty passive slave, or to temporarily improve cluster throughput and/or availability. However, sometimes, temporary solutions tend to become long term solutions. As such, client applications and their administrators would clearly benefit if active slaves could also receive the cluster context information, and be able to promote themselves to master if needed. This is an easy improvement and will be certainly developed and included in the next versions of Replic8.

Although a useful feature, currently there is no support for any type of sharding in Replic8.



## 6 Validation

This chapter documents the results of the validations performed according to the evaluation points described in Evaluation sub-chapter of the Introduction.

### 6.1 Validation infrastructure

In order to test the developed solution, a cluster for System PC with four machines was set up. The cluster is composed by three laptop machines and one desktop pc running within a Local Area Network (LAN).

#### 6.1.1 Network Topology

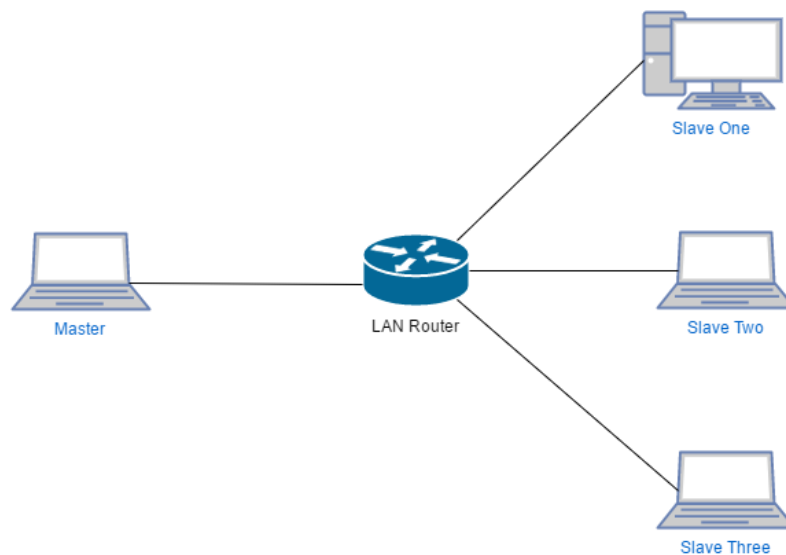


Figure 38 - Replic8 validation infrastructure setup

All machines were connected through a network router with gigabit ports.

#### 6.1.2 Hardware specifications

Table 2 - Infrastructure hardware specifications

	Operating System	CPU	Memory	Disk
Master	Ubuntu Linux 14.04 LTS	Intel Core i7-4600M @2.9Ghz x 2	16GB	Intel SSD Pro 2500 Series 240GB

Slave One	Linux Mint 18	Intel Core i5-2500K @3.3Ghz x4	8GB	Intel SSD 330 Series 120GB
Slave Two	Ubuntu Linux 14.04 LTS	Intel Core i7-4500U @1.8Ghz x 2	8GB	Samsung SSD PM851 256GB
Slave Three	Linux Mint 17.3	Intel Core i5-3317U @1.7Ghz x2	4GB	Samsung SSD 840 Evo 250GB

## 6.2 Validation scenarios

### 6.2.1 A1: Write throughput Replic8 impact

This scenario intended to confirm the following hypotheses:  $H_a$  – The write throughput of the application is not slowed down by more than 25% by the Replic8 clustering framework.

For this test, a data set containing 5000 nodes were persisted on the master instance, both with and without Replic8 activated. For this test, when Replic8 is active the cluster is composed by the Master and Slave One instances.

In order to perform this validation, an A/B scenario was performed. Throughput (req/s) measurements were taken both with Replic8 activated (A) in System PC, and with Replic8 deactivated (B) in System PC. The throughput for each scenario repetition was measured with the following formula:

$$T = \frac{TNN * 1000}{MS} \quad (2)$$

Breaking down the above formula, T is the measured throughput, TNN is the number of transacted nodes and MS is the measure time in milliseconds for the test repetition.

Each scenario was repeated thirty times. The difference in % between average throughput measurements was obtained using the following formula:

$$\Delta\% = \frac{\frac{\sum TY}{RN} \times 100}{\frac{\sum TX}{RN}} - 100 \quad (3)$$

Where  $\Delta\%$  is the percentage difference,  $\sum TX$  is the sum of the measured throughput for each repeated result in req/s with Replic8 activated,  $\sum TY$  is the sum of the measured throughput for each repeated result in req/s with Replic8 deactivated. RN is the number of repetitions for each test.

The Annex B shows the time SystemPC took to persist 5000 (TNN) nodes and the correspondent throughput with both Replic8 deactivated and activated for RN=30.

Taking the above formula, and replacing it by the obtained values, one can find that the mean slowdown % of the application when Replic8 is applied is ~3%, which is much lower than 25%.

$$\Delta\% = \frac{31,999 \times 100}{31,102} - 100 = 2,884\% \quad (4)$$

To support the confidence in the result, and, as the two data sets are sufficient large, a Student T-test for independent samples was performed comparing the mean throughput differences for each A/B scenario from Annex B, against the  $H_0$  25% throughput slowdown. The 25% percentage value (8 req/s) was calculated from the mean throughput for cluster configuration with one slave instance (959,974 / 30 = 31,999 (req/s)). The test was performed with an alpha ( $\alpha$ ) value of 0,05.

Table 3 - Parametric T test between measured mean throughput slowdown and  $H_0$  value

T Test: Two Independent Samples									
SUMMARY			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Throughput difference	30	0,897	0,098927						
Maximum allowed slowdown	30	8,000	0						
Pooled			0,049464	31,9358					
T TEST: Equal Variances				Alpha	0,05				
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	0,057425	123,6868	58	2,85E-72	1,671553			yes	0,99811
Two Tail	0,057425	123,6868	58	5,7E-72	2,001717	-7,21761	-6,98771	yes	0,99811
T TEST: Unequal Variances				Alpha	0,05				
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	0,057425	123,6868	29	2,41E-41	1,699127			yes	0,999054
Two Tail	0,057425	123,6868	29	4,82E-41	2,04523	-7,22011	-6,98522	yes	0,999054

As the p-values obtained are nearly zero,  $H_0$  can be rejected. This test successfully accepts the claim  $H_a$  that states that the write throughput of the application is not slowed down by more than 25% by the Replic8 clustering framework for an alpha ( $\alpha$ ) of 0,05.

### 6.2.2 A2: Write throughput slave instance increase impact

This scenario intends to confirm the following hypotheses:  $H_a$  – The write throughput should not be slowed down by more than 10% for each instance added to the cluster. Due to resources limitation the number of slave instances will be constrained to three.



For this test, a data set containing 5000 nodes were persisted on the master instance.

In order to perform this validation, two A/B tests were performed. Throughput (req/s) measurements were taken with cluster composition of Master and one, two and three slave instances. The first test compares the performance between a cluster composed with one slave instance (A), and with two slave instances (B). The second test compares the performance between a cluster composed with two slave instances (A) and a with three slave instances (B). The throughput for each scenario repetition was measured with the following formula:

$$T = \frac{TNN * 1000}{MS} \quad (5)$$

As in the previous test scenario, T is the measured throughput, TNN is the number of transacted nodes and MS is the measure time in milliseconds for the test repetition.

The difference in % was obtained using the following formula:

$$\Delta\% = \frac{\frac{\sum TY}{RN} \times 100}{\frac{\sum TX}{RN}} - 100 \quad (6)$$

Where  $\Delta\%$  is the percentage difference,  $\sum TY$  is the sum of the measured throughput for each repeated result in req/s with the cluster configured with one master and one slave,  $\sum TX$  is the sum of the measured throughput for each repeated result in req/s for each one of the other two cluster configurations. RN is the number of repetitions for each test.

Annex C shows the results of persisting 5000 nodes on System PC both with one and two slave instances on the cluster for a test repetition number of 30.

Using the formula with the values from Annex C results in an slowdown percentage of 0,39% when a second instance is added to the cluster:

$$\Delta\% = \frac{31,102 \times 100}{30,981} - 100 = 0,39\% \quad (7)$$

In order to rule out hypotheses  $H_0$ , a Student T-test for independent samples was performed comparing the mean throughput differences for each A/B scenario from Annex C against the  $H_0$  10% slowdown claim. The 10% percentage value (3,110 (req/s)) was calculated from the mean throughput for cluster configuration with one slave instance (933,054/ 30 = 3,110 (req/s)). The test was performed with an alpha ( $\alpha$ ) value of 0,05.

Table 4 - T test between the measured mean throughput slowdown with two slaves and  $H_0$  value

T Test: Two Independent Samples									
SUMMARY			Hyp Mean Dif	0					
Groups	Count	Mean	Variance	Cohen d					
Throughput difference	30	0,121207	0,137596466						
Maximum allowed slowdown	30	3,110	0						
Pooled			0,068798233	11,39481369					
T TEST: Equal Variances									
	std err	t-stat	df	p-value	t-crit	lower	upper	sig	effect r
One Tail	0,067724	44,13192	58	1,2521E-46	1,671553			yes	0,985435
Two Tail	0,067724	44,13192	58	2,50421E-46	2,001717	-3,12436	-2,85323	yes	0,985435
T TEST: Unequal Variances									
	std err	t-stat	df	p-value	t-crit	lower	upper	sig	effect r
One Tail	0,067724	44,13192	29	1,91793E-28	1,699127			yes	0,992637
Two Tail	0,067724	44,13192	29	3,83586E-28	2,04523	-3,1273	-2,85028	yes	0,992637

As stated in the test, the p-value is nearly zero, which rules out  $H_0$ . As such  $H_a$ , which states that, the write throughput should not be slowed down by more than 10% for each instance added to the cluster, can be accepted for an alpha ( $\alpha$ ) value of 0,05 when a second slave instance is added to the cluster.

In Annex D are the results of persisting 5000 nodes on System PC both with two and three slave instances on the cluster for a test repetition number of 30.

Using the formula with the documented values in Annex D results in an slowdown percentage of 1,75% when a third instance is added to the cluster:

$$\Delta\% = \frac{30,981 \times 100}{30,448} - 100 = 1,75\% \quad (8)$$

Once again to support the confidence in the result and rule out  $H_0$ , a Student T-test for independent samples was performed comparing the mean throughput differences for each A/B scenario from Annex D against the  $H_0$  10% slowdown claim. The 10% percentage value (3,110 (req/s)) as stated in the previous test, represents 10% of the mean time for a single transaction with the cluster configured with one instance. The test was performed with an alpha ( $\alpha$ ) value of 0,05.

Table 5- T test between the measured mean transaction slowdown with three slaves and  $H_0$  value

T Test: Two Independent Samples									
SUMMARY			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Throughput difference	30	0,532	0,084385						
Maximum allowed slowdown	30	3,110	0						
Pooled			0,042193	12,54855					
T TEST: Equal Variances				Alpha	0,05				
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	0,053036	48,60034	58	5,39E-49	1,671553			yes	0,987944
Two Tail	0,053036	48,60034	58	1,08E-48	2,001717	-2,68374	-2,47142	yes	0,987944
T TEST: Unequal Variances				Alpha	0,05				
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	0,053036	48,60034	29	1,21E-29	1,699127			yes	0,993917
Two Tail	0,053036	48,60034	29	2,43E-29	2,04523	-2,68605	-2,46911	yes	0,993917

As the p-value is nearly zero,  $H_0$  can be rejected for an alpha value of 0,05. As such  $H_a$ , which states that, the write throughput should not be slowed down by more than 10% for each instance added to the cluster, can be accepted for an alpha ( $\alpha$ ) value of 0,05 when a third slave instance is added to the cluster.

### 6.2.3 B1: Evaluation of the final state of all the database instances

In this test scenario, the objective is to confirm  $H_a$  which claims that after a test run, all instances will be consistent with each other regarding the persisted data.

The verification for this test scenario was performed when measurements for scenario **A2** were taken. For each cluster instance, including the master, notes were taken at the end of each test repetition in order to both get the final persisted version (PV) recorder by Replic8 and the number of nodes persisted in Neo4J database. Annex E shows the final results for each test run when the cluster was running with three slave instances.

From the gathered results, it can be verified that every slave instances become consistent with the master, both regarding the reported Replic8 local persistence version and also on Neo4J database. Therefore  $H_0$  can be rejected with some confidence and therefore  $H_a$  which states that 'After a test run, all instances will be consistent with each other regarding the persisted data' can be confirmed.

#### 6.2.4 B2: Slave persistence convergence time

This scenario intends to confirm hypothesis  $H_a$  which states that, after a test run, all instances should converge to a consistent state with the master instance within a time frame representing 25% of total test run time.

As with scenario B1, the data needed to verify this scenario were collected when measuring the performance for scenario A2. Measurements were made for cluster configurations containing two and three slave instances where a data set of 5000 nodes was persisted on the master instance.

This validation was performed using two A/B tests. Convergence time (mm:ss) measurements were taken with cluster composition of a Master, two and three slave instances. The first test compares the global transaction time on the master instance (A) with the convergence time for each slave instance for a cluster composed with two slaves instances (B). The second test compares the global transaction time on the master instance (A) with the convergence time for each slave instance for a cluster composed with three slaves instances (B).

The difference in % was obtained using the following formula:

$$\Delta\% = \frac{\frac{\sum CTY}{RN} \times 100}{\frac{\sum OTX}{RN}} - 100 \quad (9)$$

Where  $\Delta\%$  is the percentage difference,  $\sum CTY$  is the sum of the measured convergence time for each slave repeated result in ms,  $\sum OTX$  is the sum of the measured global operation time for each repeated result in ms for the master instance. RN is the number of repetitions for each test.

Annex F shows repeated test scenario times for master operation time and Slave One and Slave Two convergence times, both in hh:mm:ss format and milliseconds.

Taking the formula with the gathered values for the Slave One results in a convergence time difference against the master instance of 1,76%.

$$\Delta\% = \frac{164239,200 \times 100}{161400,693} - 100 = 1,76\% \quad (10)$$

Applying the same formula for the values gathered for the Slave Two, the time difference to the master instance is 0,93%

$$\Delta\% = \frac{162897,767 \times 100}{161400,693} - 100 = 0,93\% \quad (11)$$

To support the confidence in the results, and, as once again the two data sets are sufficient large, a Student T-test for independent samples was performed comparing the convergence time differences for each A/B scenario from Annex F against the  $H_0$  25% convergence time. The 25% percentage value (40350,173 ms) was calculated from the mean operation time in ms for the master instance (4842020.800 / 30 = 161400,693 (ms)). The test was performed with an alpha ( $\alpha$ ) value of 0,05.

Table 6 - Slave One convergence times against  $H_0$  limit value for a two slave cluster

T Test: Two Independent Samples									
SUMMARY			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Time difference	30	2838,507	30874587						
Maximum Time	30	40350,173	0						
Pooled			15437294	9,547305					
T TEST: Equal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1014,472	36,9765516	58	2,53E-42	1,671553			yes	0,979442
Two Tail	1014,472	36,9765516	58	5,05E-42	2,001717	-39542,4	-35481	yes	0,979442
T TEST: Unequal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1014,472	36,9765516	29	2,97E-26	1,699127			yes	0,989561
Two Tail	1014,472	36,9765516	29	5,94E-26	2,04523	-39586,5	-35436,8	yes	0,989561

Table 7 - Slave Two convergence times against  $H_0$  limit value for a two slave cluster

T Test: Two Independent Samples									
SUMMARY			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Time difference	30	1497,073	25718352						
Maximum Time difference	30	40350,173	0						
Pooled			12859176	10,83476					
T TEST: Equal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	925,8933	41,9628265	58	2,13E-45	1,671553			yes	0,983927
Two Tail	925,8933	41,9628265	58	4,27E-45	2,001717	-40706,5	-36999,7	yes	0,983927
T TEST: Unequal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	925,8933	41,9628265	29	8,09E-28	1,699127			yes	0,991866
Two Tail	925,8933	41,9628265	29	1,62E-27	2,04523	-40746,8	-36959,4	yes	0,991866

As both Table 6 and Table 7 shows the p-value is nearly zero for each case.  $H_0$  can be rejected for an alpha value of 0,05. As such  $H_a$ , can be accepted for an alpha ( $\alpha$ ) value of 0,05 for a cluster configuration with a Master and two slaves.

The Annex G shows the convergence times for slave instances in a cluster composed of a master instance and three slave instances.

Using the formula with measured values for the Slave One, results in a convergence time difference against the master instance of 10,53%.

$$\Delta\% = \frac{178406,767 \times 100}{161400,693} - 100 = 10,53\% \quad (12)$$

Applying the same formula for the values gathered for the Slave Two, the time difference to the master instance is 8,61%

$$\Delta\% = \frac{175301,733 \times 100}{161400,693} - 100 = 8,61\% \quad (13)$$

Using again the above formula for the values gathered for the Slave Three, results in a time difference to the master instance of 9,16%

$$\% = \frac{176183,067 \times 100}{161400,693} - 100 = 9,16\% \quad (14)$$

As with the previous tests, in order support the confidence in the results, a Student T-test for independent samples was performed comparing the convergence time differences for each A/B scenario from Annex G against the  $H_0$  25% maximum convergence time. As stated before, the 25% percentage value (40350,173 ms) was calculated from the mean operation time in ms for the master instance. As the previous tests, each test was performed with an alpha ( $\alpha$ ) value of 0,05.

Table 8 - Slave One convergence times against  $H_0$  limit value for a three slave cluster

T Test: Two Independent Samples									
SUMMARY			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Time	30	17006,073	83192919						
Maximum Time difference	30	40350,173	0						
Pooled			41596460	3,619503					
T TEST: Equal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1665,3	14,0182752	58	1,4E-20	1,671553			yes	0,8787
Two Tail	1665,3	14,0182752	58	2,79E-20	2,001717	-26677,5	-20010,7	yes	0,8787
T TEST: Unequal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1665,3	14,0182752	29	9,5E-15	1,699127			yes	0,93349
Two Tail	1665,3	14,0182752	29	1,9E-14	2,04523	-26749,9	-19938,3	yes	0,93349

Table 9 - Slave Two convergence times against  $H_0$  limit value e for a three slave cluster

T Test: Two Independent Samples									
SUMMARY			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Time difference	30	13901,040	83071603						
Maximum Time difference	30	40350,173	0						
Pooled			41535802	4,103932					
T TEST: Equal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1664	15,89446	58	4,21E-23	1,671553			yes	0,901823
Two Tail	1664	15,89446	58	8,43E-23	2,001717	-29780,1	-23118,2	yes	0,901823
T TEST: Unequal Variances					Alpha	0,05			
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1664	15,89446	29	3,74E-16	1,699127			yes	0,947116
Two Tail	1664	15,89446	29	7,47E-16	2,04523	-29852,5	-23045,8	yes	0,947116

Table 10 - Slave Three convergence times against  $H_0$  limit value e for a three slave cluster

T Test: Two Independent Samples									
SUMMARY									
			Hyp Mean	0					
<i>Groups</i>	<i>Count</i>	<i>Mean</i>	<i>Variance</i>	<i>Cohen d</i>					
Time difference	30	13901,040	83071603						
Maximum Time difference	30	40350,173	0						
Pooled			41535802	4,103932					
T TEST: Equal Variances									
			Alpha	0,05					
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1664	15,89446	58	4,21E-23	1,671553			yes	0,901823
Two Tail	1664	15,89446	58	8,43E-23	2,001717	-29780,1	-23118,2	yes	0,901823
T TEST: Unequal Variances									
			Alpha	0,05					
	<i>std err</i>	<i>t-stat</i>	<i>df</i>	<i>p-value</i>	<i>t-crit</i>	<i>lower</i>	<i>upper</i>	<i>sig</i>	<i>effect r</i>
One Tail	1664	15,89446	29	3,74E-16	1,699127			yes	0,947116
Two Tail	1664	15,89446	29	7,47E-16	2,04523	-29852,5	-23045,8	yes	0,947116

As the three Student T tests show, for an alpha ( $\alpha$ ) value of 0,05,  $H_0$  can be rejected as the p values are close to zero. Therefore,  $H_a$  can be accepted for an alpha ( $\alpha$ ) value of 0,05 for a cluster configuration with a Master and two three slaves.

Although, a cluster configuration with 4 slaves would probably short the gap to the 25% convergence time threshold, for the time being there are no plans on having more than 3 slave instances of System PC running at the same time. Also Replic8, is on a prototype state and not fully optimized yet which can justify the performance hit with the 3 slave configuration.

### 6.2.5 C1: Cluster behavior when a slave instance goes offline

This scenario aims to verify the claim  $H_a$  which states that when one or more slave instances are shutdown, the cluster continues behave the same way, handling transaction replication successfully. All the remaining online instances should continue to receive and process transactions. Also, when the failing slave comes back online again, it should re-sync with the master instance in order to achieve persistence convergence.

For this scenario, three tests were performed, one disconnecting one slave, another disconnecting two slaves, and the last one disconnecting all the three slaves. For all the tests, the slave is brought back online in active mode in order to register itself within the cluster and request the missing transactions.



```

Transaction was committed. Will now update the Transaction Recovery Log and broadcast it to the cluster.
2017-04-13 00:57:12,970 INFO ptc p.r.r.l.t.l.TransactionRecoveryLogger [RMI TCP Connection(4)-127.0.0.1] INFO :
Updating Transaction Recovery Logger to version [464]
2017-04-13 00:57:12,979 INFO ptc p.r.r.l.t.i.AbstractTransactionInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
Evaluating transaction result ...
2017-04-13 00:57:12,991 INFO ptc p.a.a.s.g.s.c.i.SkyWalkerCacheInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
SkyWalker Cache Interceptor: Evaluating response...
2017-04-13 00:57:12,992 INFO ptc p.a.a.s.g.s.c.i.SkyWalkerCacheInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
New update on graph database. Cache will be flushed.
2017-04-13 00:57:12,992 INFO ptc p.r.r.l.t.i.AbstractTransactionInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
Transaction was committed. Will now update the Transaction Recovery Log and broadcast it to the cluster.
2017-04-13 00:57:12,992 INFO ptc p.r.r.l.t.l.TransactionRecoveryLogger [RMI TCP Connection(4)-127.0.0.1] INFO :
Updating Transaction Recovery Logger to version [465]
2017-04-13 00:57:12,998 WARN ptc p.r.r.m.b.o.ErlangTransactionObserver [pool-8-thread-1] WARN :
The slave instance slave one@lpt seems to be offline. The correspondent observer will be unregistered.
2017-04-13 00:57:12,999 INFO ptc p.r.r.l.b.TransactionBroadcaster [pool-8-thread-1] INFO :
Unregistered observer for instance: [slave one@lpt]
2017-04-13 00:57:13,002 INFO ptc p.r.r.l.t.i.AbstractTransactionInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
Evaluating transaction result ...
2017-04-13 00:57:13,017 INFO ptc p.a.a.s.g.s.c.i.SkyWalkerCacheInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
SkyWalker Cache Interceptor: Evaluating response...
2017-04-13 00:57:13,017 INFO ptc p.a.a.s.g.s.c.i.SkyWalkerCacheInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
New update on graph database. Cache will be flushed.
2017-04-13 00:57:13,017 INFO ptc p.r.r.l.t.i.AbstractTransactionInterceptor [RMI TCP Connection(4)-127.0.0.1] INFO :
Transaction was committed. Will now update the Transaction Recovery Log and broadcast it to the cluster.
2017-04-13 00:57:13,018 INFO ptc p.r.r.l.t.l.TransactionRecoveryLogger [RMI TCP Connection(4)-127.0.0.1] INFO :
Updating Transaction Recovery Logger to version [466]

```

Figure 39 - Slave disconnection from the cluster

During each test it was observed that the cluster remained unaffected when one or more slaves were disconnected. As can be seen in Figure 39, the transaction broadcast flow continued as usual for the remaining online slaves. Also, each slave that went offline successfully registered itself within the cluster and its persistence version converged with the master instance after being brought back online.

From these results, it can be seen that the master instance continued to work properly, the remaining online slaves also continued to receive transactions. As such  $H_0$  can be rejected with confidence, and  $H_a$  can be confirmed.

## 6.2.6 C2: Cluster behavior when the master instance goes offline

The objective of this scenario is to verify that the claim  $H_a$  which states that after shutting down the master instance, the failover is successfully performed and another instance immediately assumes the master role, ensuring the cluster remain available and accepting writes. To test this scenario, the master instance was shut down, and it was verified if the failover to the next slave in the hierarchy was processed successfully. This test was repeated three times.

```

Loaded Slave Three Instance cluster configuration.
2017-04-18 21:19:03,085 INFO LinuxPowerThesis p.r.r.m.c.s.e.ErlangSender [pool-8-thread-1] INFO :
An message was sent to master@s440 for {#Pid<slave_one@lpt.2.0>,check_alive}
2017-04-18 21:19:05,088 WARN LinuxPowerThesis p.r.r.m.c.h.SlaveHealthCheckService [DefaultQuartzScheduler_Worker-1] W
ARN :
Assuming Master role as the previous Master instance has become unreachable
2017-04-18 21:19:05,088 INFO LinuxPowerThesis p.r.r.l.c.r.e.ErlangReceiver [DefaultQuartzScheduler_Worker-1] INFO :
Shutting down transaction service Erlang Service...
2017-04-18 21:19:05,089 INFO LinuxPowerThesis p.r.r.l.c.r.e.ErlangReceiver [DefaultQuartzScheduler_Worker-1] INFO :
transaction service Erlang Service successfully shut down
2017-04-18 21:19:05,089 INFO LinuxPowerThesis p.r.r.l.c.r.e.ErlangReceiver [DefaultQuartzScheduler_Worker-1] INFO :
Shutting down cluster_context service Erlang Service...
2017-04-18 21:19:05,118 INFO LinuxPowerThesis p.r.r.l.c.r.e.ErlangReceiver [DefaultQuartzScheduler_Worker-1] INFO :
cluster_context service Erlang Service successfully shut down
2017-04-18 21:19:05,178 INFO LinuxPowerThesis p.r.r.l.c.i.c.w.ClusterInstanceDefinitionsConfigWriter [DefaultQuartzSc
heduler_Worker-1] INFO :
Going to save the cluster instance definitions.
2017-04-18 21:19:05,202 INFO LinuxPowerThesis p.r.r.l.c.i.c.w.ClusterInstanceDefinitionsConfigWriter [DefaultQuartzSc
heduler_Worker-1] INFO :
Cluster instance definitions saved.
2017-04-18 21:19:05,204 INFO LinuxPowerThesis p.r.r.l.c.i.c.w.MasterInitializer [DefaultQuartzScheduler_Worker-1] INFO :
Starting Replic8 Clustering as a [MASTER] instance.
2017-04-18 21:19:05,204 INFO LinuxPowerThesis p.r.r.l.c.i.c.l.ClusterInstanceDefinitionsConfigLoader [DefaultQuartzSc
heduler_Worker-1] INFO :
Loading cluster instance configurations...
2017-04-18 21:19:05,210 INFO LinuxPowerThesis p.r.r.l.c.i.c.l.ClusterInstanceDefinitionsConfigLoader [DefaultQuartzSc
heduler_Worker-1] INFO :
Loaded Slave Three Instance cluster configuration.
2017-04-18 21:19:05,210 INFO LinuxPowerThesis p.r.r.l.c.i.c.l.ClusterInstanceDefinitionsConfigLoader [DefaultQuartzSc
heduler_Worker-1] INFO :
Loaded Slave One Instance cluster configuration.
2017-04-18 21:19:05,223 INFO LinuxPowerThesis p.r.r.l.c.r.e.ErlangReceiver [DefaultQuartzScheduler_Worker-1] INFO :
Starting up registry service Erlang Service...
2017-04-18 21:19:05,223 INFO LinuxPowerThesis p.r.r.l.c.r.e.ErlangReceiver [DefaultQuartzScheduler_Worker-1] INFO :
registry service Erlang Service successfully started
2017-04-18 21:19:05,225 INFO LinuxPowerThesis p.r.r.m.c.i.ClusterInitializer [DefaultQuartzScheduler_Worker-1] INFO :
Starting the cluster...
2017-04-18 21:19:05,229 INFO LinuxPowerThesis p.r.r.m.c.i.ClusterInitializer [DefaultQuartzScheduler_Worker-1] INFO :
Connecting with instance [Slave Three Instance] with address [slave_three@s9].
2017-04-18 21:19:08,232 INFO LinuxPowerThesis p.r.r.m.c.s.e.ErlangSender [pool-9-thread-1] INFO :
An message was sent to slave three@s9 for {#Pid<slave_one@lpt.3.0>,{cluster_context,{"SLAVE","Sla
ve Three Instance"},"slave_three@s9"},"MASTER","Slave One Instance","slave_one@lpt"}}

```

Figure 40 - Master Failover to next in hierarchy slave

The test performed successfully confirms that the failover was successfully executed to the next slave in the hierarchy when the master instance goes offline. It was also verified that the new master informed the remaining slave instances about the new cluster composition.

These results show that the master failover was successfully handled by the next in the hierarchy slave instance. As such,  $H_0$  can be rejected with confidence, and  $H_a$  can be confirmed.

## 6.3 Conclusion

The test scenarios and the results gathered in this chapter played a very important role, both to assert the functionality and the quality attributes of the framework. During this phase some minor coding problems were identified and corrected.

The results obtained for each scenario, shows that Replic8 is compliant with the prerequisites established both in Evaluation and Functional Requirements chapters.



## 7 Conclusions

This master thesis, and the work described in it, aimed to address a real world problem, adding cluster capabilities to an application that is already in production, serving thousands of requests per day, without resorting to any kind of commercial solution.

In order to solve the problem in hand, a thoughtfully study both on the major database replication concepts, its advantages and pitfalls, and the state of the art regarding existing replication frameworks has been performed, evaluating their ability to solve the exposed problem. The knowledge and information gathered allowed the development of Replic8, a replication framework flexible enough to be applied in any java backend application regardless its persistence type, while addressing all the requirements that led to its development.

Replic8 was developed following the more relaxed BASE principle, gearing it towards the Availability and Partition Tolerant ends of the CAP theorem in trade for consistency. The work presented, not only addresses the original problem, but can also serve as a good base to be further developed, and used by others, as its implementation is not bound in any way to the application that led to its development.

The development of this master thesis not only resulted in the contribution of a new replication framework, but also played an important role providing very useful knowledge on key replication concepts that will be more and more important in the future with the proliferation of distributed cloud based infrastructures.

Through this work, a clear identification of the goals and quality attributes for the replication framework to be developed were written. These goals assured that the developed framework would successfully address the problem in hand, and were later validated when Replic8 was put to test.

The study performed on existing replication frameworks was very helpful, not only to evaluate the ability of those frameworks to solve the exposed problem, but also to learn how these frameworks addressed common replication problems.

The design and development of Replic8 posed a considerable technical challenge that in the end was successfully overcome. The biggest challenge was to design a solution that would work for any type of database the application interacts with. Although the objective was achieved, it imposed some tradeoffs.

The validation of the framework was performed following a set of pre-established scenarios derived from the identified goals and quality attributes. Replic8 successfully passed in all scenarios, which assures the compliance with its functional requirements. Nevertheless, during this process, some limitations and technical improvements were identified, and should be addressed in future developments.

## 7.1 Open Issues and Future Work

Although the main objectives for the proposed solution in this document were successfully achieved, there are certainly some improvements left to be done. Chapter 5.6 (Limitations), already identifies and exposes some of those issues. The following paragraphs add some considerations about a few identified technical improvements that would improve Replic8 performance and usability.

Currently there is no rotation policy for the recovery logger file. This means that this file will grow over time, deteriorating both read and write times. A possible approach is to create a new file each N number of transactions. This way both reads and writes would be redirected to a much smaller file, improving the correspondent operation performance.

Add support to include statefull java beans context along with the transaction itself is an important task as it would wide up the possible applications for Replic8.

Currently, the major open issue in Replic8 is how to recover from a master instance failure that has successfully committed a transaction to the database and gone offline before writing it to the recovery log file and send it to the remainder cluster instances. The resolution for this issue may involve including the writing to the recovery log file and broadcasting the transaction at least to the next in line slave instance inside the same transaction as the commit to the database.

As future work, in order to help managing the cluster and monitoring its health, Replic8 would benefit from having a Front End application to carry on such tasks. This application would provide tools for cluster instances configuration, cluster health monitoring and also some statistic metrics about the cluster activity.

A proper software licensing will also be studied and applied in the future in order to better manage the distribution of the Replic8 clustering framework.

# References

- Allee, V., 2008. Value network analysis and value conversion of tangible and intangible assets. *Journal of Intellectual Capital*, 9(1), pp.5-24. Available at: <http://dx.doi.org/10.1108/14691930810845777>.
- Amazon, n.d. *Amazon Web Services - Cloud Computing Services*. [Online] Available at: <https://aws.amazon.com/> [Accessed 13 February 2016].
- Amazon, n.d. *AWS | Amazon Relational Database Service*. [Online] Available at: <https://aws.amazon.com/rds/> [Accessed 13 February 2016].
- ANSI X3.135-1992, 1992. American National Standard for Information Systems -- Database Language -- SQL.
- Armstrong, J., Viriding, R., Wikstr & Williams, M., 1993. Concurrent programming in ERLANG.
- Berenson, H. et al., 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, 1995. ACM.
- Bernstein, P.A., Hadzilacos, V. & Goodman, N., 1987. *Concurrency control and recovery in database systems*. Addison-wesley New York.
- Brebner, P. & Gosper, J., 2003. How Scalable is J2EE Technology? *SIGSOFT Softw. Eng. Notes*, 28(3), pp.4-4. Available at: <http://doi.acm.org/10.1145/773126.773139>.
- Brewer, E.A., 2000. Towards robust distributed systems. In *PODC.*, 2000.
- Brewer, E., 2012. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2), pp.23-29.
- Carnevale, P.J. & Pruitt, D.G., 1992. Negotiation and mediation. *Annual review of psychology*, 43(1), pp.531-82.
- Cecchet, E., Candea, G. & Ailamaki, A., 2008. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA, 2008. ACM.
- Cecchet, E., Marguerite, J. & Zwaenepole, W., 2004. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA, 2004. USENIX Association.
- Chen, S. et al., 2011. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Rec.*, 39(3), pp.5-10. Available at: <http://doi.acm.org/10.1145/1942776.1942778>.

- Coulouris, G.F., Dollimore, J. & Kindberg, T., 2005. *Distributed systems: concepts and design*. pearson education.
- Daudjee, K. & Salem, K., 2006. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases.*, 2006.
- Elnikety, S., Dropsho, S. & Pedone, F., 2006. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *ACM SIGOPS Operating Systems Review.*, 2006.
- Filzmoser, M. & Vetschera, R., 2008. A classification of bargaining steps and their impact on negotiation outcomes. *Group Decision and Negotiation*, 17(5), pp.421-43.
- Fox, A. & Brewer, E.A., 1999. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on.*, 1999.
- Fox, A. et al., 1997. *Cluster-Based Scalable Network Services*. ACM.
- Gamma, E., 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
- Gilbert, S. & Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), pp.51-59.
- Gilbert, S. & Lynch, N.A., 2012. Perspectives on the CAP Theorem., 2012.
- Google, n.d. *AngularJS - Superheroic JavaScript MVW Framework*. [Online] Available at: <https://angularjs.org/> [Accessed 12 February 2016].
- Gray, J., 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. London, UK, UK, 1978. Springer-Verlag.
- Gray, J., Helland, P., O'Neil, P. & Shasha, D., 1996. The Dangers of Replication and a Solution. *SIGMOD Rec.*, 25(2), pp.173-82. Available at: <http://doi.acm.org/10.1145/235968.233330>.
- Gray, J.N., Lorie, R.A., Putzolu, G.R. & Traiger, I.L., 1976. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems.*, 1976.
- Gray, J. & Siewiorek, D.P., 1991. High-availability computer systems. *Computer*, 24(9), pp.39-48.
- Haerder, T. & Reuter, A., 1983. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4), pp.287-317. Available at: <http://doi.acm.org/10.1145/289.291>.
- HAProxy, n.d. *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer*. [Online] Available at: <http://www.haproxy.org/> [Accessed 13 February 2016].
- Helland, P., 2007. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR.*, 2007.

Koen, P.A. et al., 2002. *Fuzzy front end: effective methods, tools, and techniques*. Wiley, New York, NY.

Koen, P. et al., 2001. Providing clarity and a common language to the “fuzzy front end”. *Research-Technology Management*, 44(2), pp.46-55.

Laddad, R., 2003. *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press.

Lieberman, H.J. et al., 1997. Negotiating barriers to intensive case management: The triple win model. *Administration and Policy in Mental Health and Mental Health Services Research*, 24(3), pp.251-56.

Liu, L. & Zsu, M.T., 2009. *Encyclopedia of Database Systems*. 1st ed. Springer Publishing Company, Incorporated.

Management Study Guide, n.d. *Models of Negotiation*. [Online] Available at: <http://www.managementstudyguide.com/models-of-negotiation.htm> [Accessed 14 February 2016].

Marcus, E. & Stern, H., 2003. *Blueprints for high availability*. John Wiley & Sons.

Michael, M., Moreira, J.E., Shiloach, D. & Wisniewski, R.W., 2007. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International.*, 2007.

Montag, D., 2013. Understanding neo4j scalability. *White Paper, Neotechnology*.

Neo4J, n.d. *Neo4j Graph Database*. [Online] Available at: <http://neo4j.com/product/> [Accessed 12 February 2016].

Nicola, S., 2015. [Online] Available at: [https://moodle.isep.ipp.pt/pluginfile.php/91647/mod\\_resource/content/2/An%C3%A1lise\\_Va\\_lor\\_Aula1.pdf](https://moodle.isep.ipp.pt/pluginfile.php/91647/mod_resource/content/2/An%C3%A1lise_Va_lor_Aula1.pdf) [Accessed 12 February 2016].

Nicola, S., Ferreira, E.P. & Ferreira, J.J.P., 2012. A novel framework for modeling value for the customer, an essay on negotiation. *International Journal of Information Technology & Decision Making*; Vol. 11, Issue 3, 11(03), pp.661-703.

Nicola, S., Ferreira, E.P. & Ferreira, J.J.P., 2014. A Quantitative Model for Decomposing & Assessing the Value for the Customer. *Journal of Innovation Management*, 2(1), pp.104-38.

Osterwalder, A., 2004. The business model ontology: A proposition in a design science approach.

Ozsu, M.T., 2007. *Principles of Distributed Database Systems*. 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press.



Pivotal Software, n.d. *RabbitMQ - Messaging that just works*. [Online] Available at: <https://www.rabbitmq.com/> [Accessed 13 February 2016].

Pivotal Software, n.d. *Spring Framework*. [Online] Available at: <https://projects.spring.io/spring-framework/> [Accessed 12 February 2016].

Plattner, C. & Alonso, G., 2004. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware.*, 2004.

Poess, M. & Floyd, C., 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.*, 29(4), pp.64-71. Available at: <http://doi.acm.org/10.1145/369275.369291>.

PostgreSQL, n.d. *The world's most advanced open source database*. [Online] Available at: [www.postgresql.org](http://www.postgresql.org) [Accessed 12 February 2016].

Pritchett, D., 2008. Base: An acid alternative. *Queue*, 6(3), pp.48-55.

Saito, Y. & Shapiro, M., 2005. Optimistic Replication. *ACM Comput. Surv.*, 37(1), pp.42-81. Available at: <http://doi.acm.org/10.1145/1057977.1057980>.

Tanenbaum, A.S. & Van Steen, M., 2002. *Distributed systems: principles and paradigms*. Prentice hall Englewood Cliffs.

TIOBE Software BV, n.d. *TIOBE - TIOBE Index*. [Online] Available at: <http://www.tiobe.com/tiobe-index/> [Accessed 28 December 2016].

Woodall, T., 2003. Conceptualising 'value for the customer': an attributional, structural and dispositional analysis. *Academy of marketing science review*, 2003, p.1.

Zeithaml, V.A., 1988. Consumer perceptions of price, quality, and value: a means-end model and synthesis of evidence. *The Journal of marketing*, pp.2-22.

## Annex A – Canvas Model

The following figure represents the Canvas Model described in Value Analysis.

<u><b>Key Partners</b></u> Oracle Pivotal Open-source Community	<u><b>Key Activities</b></u> Research Software Architecture & Design Software Development Software Testing Support	<u><b>Value Propositions</b></u> Increase Availability Increase performance Database Agnostic No change to existing DB schemas	<u><b>Customer Relationships</b></u> Self-service Software development community Improvements requests Support	<u><b>Customer Segments</b></u> Software Developers Software Development Companies
	<u><b>Key Resources</b></u> Software Developers Workstations Software Installations		<u><b>Channels</b></u> Project Web Site Github.com Maven Central Forked Software	
<u><b>Cost Structure</b></u> Research & Development Installations		<u><b>Revenue Streams</b></u> Free version (Community Edition) Paid version (Enterprise Edition)		

Figure 41 - Canvas Model

## Annex B - A/B test throughput for SystemPC w/o and with Replic8

Table 11 - A/B test throughput for SystemPC w/o and with Replic8

Repetition	Time W/O Replic8	Time with Replic8	Throughput w/o R8 (req/s)	Throughput with R8 (req/s)
1	00:02:37,320	00:02:39,352	31,782	31,377
2	00:02:37,802	00:02:41,702	31,685	30,921
3	00:02:36,638	00:02:43,658	31,921	30,552
4	00:02:34,550	00:02:38,260	32,352	31,594
5	00:02:35,464	00:02:40,986	32,162	31,059
6	00:02:36,738	00:02:40,886	31,900	31,078
7	00:02:37,213	00:02:40,666	31,804	31,120
8	00:02:36,395	00:02:41,366	31,970	30,985
9	00:02:37,118	00:02:41,799	31,823	30,903
10	00:02:35,719	00:02:39,012	32,109	31,444
11	00:02:36,108	00:02:43,281	32,029	30,622
12	00:02:36,731	00:02:40,101	31,902	31,230
13	00:02:36,399	00:02:40,096	31,970	31,231
14	00:02:35,029	00:02:41,285	32,252	31,001
15	00:02:35,925	00:02:41,309	32,067	30,996
16	00:02:35,519	00:02:42,099	32,150	30,845
17	00:02:34,997	00:02:39,348	32,259	31,378
18	00:02:34,596	00:02:41,369	32,342	30,985
19	00:02:35,283	00:02:40,101	32,199	31,230
20	00:02:35,109	00:02:39,547	32,235	31,339
21	00:02:37,455	00:02:41,090	31,755	31,056
22	00:02:37,386	00:02:38,491	31,769	31,548
23	00:02:37,277	00:02:40,429	31,791	31,166
24	00:02:37,797	00:02:42,362	31,686	30,795
25	00:02:35,024	00:02:42,373	32,253	30,793
26	00:02:35,712	00:02:41,010	32,111	31,056
27	00:02:36,144	00:02:39,253	32,022	31,397
28	00:02:37,021	00:02:40,807	31,843	31,093
29	00:02:35,659	00:02:41,571	32,121	30,946
30	00:02:37,688	00:02:39,679	31,708	31,313

## Annex C - A/B test throughput for Replic8 with one and two slaves

Table 12 - A/B test throughput for Replic8 with one and two slaves

Repetition	Time W/O Replic8	Time with Replic8	Throughput with one slave (req/s)	Throughput with two slaves (req/s)
1	00:02:39,352	00:02:41,138	31,377	31,029
2	00:02:41,702	00:02:43,429	30,921	30,594
3	00:02:43,658	00:02:41,295	30,552	30,999
4	00:02:38,260	00:02:41,495	31,594	30,961
5	00:02:40,986	00:02:40,558	31,059	31,141
6	00:02:40,886	00:02:41,543	31,078	30,952
7	00:02:40,666	00:02:40,743	31,120	31,106
8	00:02:41,366	00:02:42,027	30,985	30,859
9	00:02:41,799	00:02:41,513	30,903	30,957
10	00:02:39,012	00:02:39,794	31,444	31,290
11	00:02:43,281	00:02:41,856	30,622	30,892
12	00:02:40,101	00:02:42,303	31,230	30,807
13	00:02:40,096	00:02:41,684	31,231	30,925
14	00:02:41,285	00:02:42,139	31,001	30,838
15	00:02:41,309	00:02:40,39	30,996	31,250
16	00:02:42,099	00:02:40,787	30,845	31,097
17	00:02:39,348	00:02:44,276	31,378	30,437
18	00:02:41,369	00:02:41,875	30,985	30,888
19	00:02:40,101	00:02:39,81	31,230	31,446
20	00:02:39,547	00:02:40,412	31,339	31,170
21	00:02:41,09	00:02:43,692	31,056	30,545
22	00:02:38,491	00:02:42,086	31,548	30,848
23	00:02:40,429	00:02:39,867	31,166	31,276
24	00:02:42,362	00:02:40,6	30,795	31,250
25	00:02:42,373	00:02:39,973	30,793	31,255
26	00:02:41,01	00:02:41,167	31,056	31,024
27	00:02:39,253	00:02:42,928	31,397	30,688
28	00:02:40,807	00:02:40,448	31,093	31,163
29	00:02:41,571	00:02:41,152	30,946	31,027
30	00:02:39,679	00:02:42,839	31,313	30,705

## Annex D - A/B test throughput for Replic8 with two and three slaves

Table 13 - A/B test throughput for Replic8 with two and three slaves

Repetition	Time with two slaves	Time with three slaves	Throughput with two slaves (req/s)	Throughput with three slaves (req/s)
1	00:02:41,138	00:02:45,122	31,029	30,281
2	00:02:43,429	00:02:44,829	30,594	30,334
3	00:02:41,295	00:02:42,694	30,999	30,733
4	00:02:41,495	00:02:43,557	30,961	30,570
5	00:02:40,558	00:02:45,200	31,141	30,302
6	00:02:41,543	00:02:45,444	30,952	30,222
7	00:02:40,743	00:02:44,886	31,106	30,324
8	00:02:42,027	00:02:45,611	30,859	30,191
9	00:02:41,513	00:02:42,896	30,957	30,694
10	00:02:39,794	00:02:43,790	31,290	30,675
11	00:02:41,856	00:02:45,763	30,892	30,164
12	00:02:42,303	00:02:44,814	30,807	30,337
13	00:02:41,684	00:02:44,214	30,925	30,448
14	00:02:42,139	00:02:44,555	30,838	30,385
15	00:02:40,390	00:02:43,831	31,250	30,519
16	00:02:40,787	00:02:43,842	31,097	30,517
17	00:02:44,276	00:02:43,362	30,437	30,607
18	00:02:41,875	00:02:45,213	30,888	30,264
19	00:02:39,810	00:02:44,726	31,446	30,353
20	00:02:40,412	00:02:42,583	31,170	30,754
21	00:02:43,692	00:02:45,387	30,545	30,232
22	00:02:42,086	00:02:43,550	30,848	30,675
23	00:02:39,867	00:02:44,200	31,276	30,487
24	00:02:40,600	00:02:44,781	31,250	30,343
25	00:02:39,973	00:02:46,048	31,255	30,112
26	00:02:41,167	00:02:42,882	31,024	30,697
27	00:02:42,928	00:02:45,812	30,688	30,155
28	00:02:40,448	00:02:42,740	31,163	30,864
29	00:02:41,152	00:02:42,330	31,027	30,864
30	00:02:42,839	00:02:44,789	30,705	30,342

# Annex E - Persistence convergence verification results

Table 14 - Persistence convergence verification results

Repetition	Master PV / Persisted Nodes	Slave One PV / Persisted Nodes	Slave TwoPV / Persisted Nodes	Slave Three PV / Persisted Nodes	Result
1	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
2	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
3	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
4	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
5	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
6	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
7	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
8	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
9	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
10	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
11	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
12	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
13	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
14	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
15	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
16	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
17	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
18	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
19	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
20	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
21	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
22	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
23	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
24	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
25	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
26	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
27	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
28	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
29	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK
30	5000 / 5000	5000 / 5000	5000 / 5000	5000 / 5000	OK

# Annex F - Persistence convergence times for a two slave cluster

Table 15 - Persistence convergence times for a two slave cluster

Repetition	Master operation time	Slave One convergence time	Slave Two convergence time	Master operation time (ms)	Slave One convergence time (ms)	Slave Two convergence time (ms)
1	00:02:41,138	00:02:44,957	00:02:37,494	161138	164957	157494
2	00:02:43,429	00:02:36,462	00:02:46,998	163429	156462	166998
3	00:02:41,295	00:02:43,164	00:02:49,138	161295	163164	169138
4	00:02:41,495	00:02:34,703	00:02:37,937	161495	154703	157937
5	00:02:40,558	00:02:39,654	00:02:47,530	160558	159654	167530
6	00:02:41,543	00:02:37,488	00:02:37,811	161543	157488	157811
7	00:02:40,743	00:02:48,144	00:02:44,471	160743	168144	164471
8	00:02:42,027	00:02:50,389	00:02:47,024	162027	170389	167024
9	00:02:41,513	00:02:50,449	00:02:47,609	161513	170449	167609
10	00:02:39,794	00:02:48,456	00:02:36,159	159794	168456	156159
11	00:02:41,856	00:02:38,927	00:02:40,171	161856	158927	160171
12	00:02:42,303	00:02:41,251	00:02:44,318	162303	161251	164318
13	00:02:41,684	00:02:37,399	00:02:35,838	161684	157399	155838
14	00:02:42,139	00:02:49,528	00:02:41,581	162139	169528	161581
15	00:02:40,390	00:02:50,964	00:02:38,149	160000	170964	158149
16	00:02:40,787	00:02:43,381	00:02:49,784	160787	163381	169784
17	00:02:44,276	00:02:45,856	00:02:38,990	164276	165856	158990
18	00:02:41,875	00:02:46,156	00:02:41,489	161875	166156	161489
19	00:02:39,810	00:02:49,350	00:02:36,512	159001	169350	156512
20	00:02:40,412	00:02:33,539	00:02:33,456	160412	153539	153456
21	00:02:43,692	00:02:41,179	00:02:50,214	163692	161179	170214
22	00:02:42,086	00:02:44,956	00:02:50,623	162086	164956	170623
23	00:02:39,867	00:02:47,071	00:02:49,920	159867	167071	169920
24	00:02:40,600	00:02:37,315	00:02:44,158	160001	157315	164158
25	00:02:39,973	00:02:44,658	00:02:40,736	159973	164658	160736
26	00:02:41,167	00:02:49,645	00:02:48,382	161167	169645	168382
27	00:02:42,928	00:02:46,276	00:02:48,081	162928	166276	168081
28	00:02:40,448	00:02:45,504	00:02:37,097	160448	165504	157097
29	00:02:41,152	00:02:50,850	00:02:38,068	161152	170850	158068
30	00:02:42,839	00:02:49,505	00:02:47,195	162839	169505	167195

# Annex G - Persistence convergence times for a three slave cluster

Table 16 - Persistence convergence times for a three slave cluster

Repetition	Master operation time	Slave One convergence time	Slave Two convergence time	Slave Three convergence time	Master operation time (ms)	Slave One convergence time (ms)	Slave Two convergence time (ms)	Slave Three convergence time (ms)
1	00:02:41,138	00:02:48,600	00:02:52,218	00:02:40,966	161138	168600	172218	160966
2	00:02:43,429	00:03:00,441	00:03:01,485	00:03:12,839	163429	180441	181485	192839
3	00:02:41,295	00:03:13,419	00:02:43,711	00:02:50,687	161295	193419	163711	170687
4	00:02:41,495	00:02:59,125	00:02:41,324	00:02:48,642	161495	179125	161324	168642
5	00:02:40,558	00:02:46,377	00:02:50,383	00:03:07,053	160558	166377	170383	187053
6	00:02:41,543	00:02:41,163	00:03:06,064	00:03:02,722	161543	161163	186064	182722
7	00:02:40,743	00:03:07,900	00:02:42,048	00:02:49,715	160743	187900	162048	169715
8	00:02:42,027	00:03:08,240	00:03:08,919	00:03:02,501	162027	188240	188919	182501
9	00:02:41,513	00:02:58,939	00:03:12,348	00:03:00,256	161513	178939	192348	180256
10	00:02:39,794	00:02:52,037	00:02:57,887	00:02:57,395	159794	172037	177887	177395
11	00:02:41,856	00:02:55,698	00:02:48,889	00:02:48,652	161856	175698	168889	168652
12	00:02:42,303	00:02:50,676	00:02:51,289	00:03:02,830	162303	170676	171289	182830
13	00:02:41,684	00:02:54,352	00:02:54,133	00:02:56,405	161684	174352	174133	176405
14	00:02:42,139	00:03:11,639	00:02:54,269	00:02:49,027	162139	191639	174269	169027
15	00:02:40,390	00:03:04,933	00:03:04,693	00:02:59,173	160000	184933	184693	179173
16	00:02:40,787	00:03:00,248	00:03:08,844	00:03:02,617	160787	180248	188844	182617
17	00:02:44,276	00:03:06,404	00:02:50,318	00:02:53,710	164276	186404	170318	173710
18	00:02:41,875	00:02:53,054	00:02:50,007	00:02:58,089	161875	173054	170007	178089
19	00:02:39,810	00:02:53,121	00:02:58,301	00:03:04,657	159001	173121	178301	184657
20	00:02:40,412	00:02:51,648	00:03:05,630	00:03:09,993	160412	171648	185630	189993
21	00:02:43,692	00:02:41,009	00:02:57,113	00:02:50,155	163692	161009	177113	170155
22	00:02:42,086	00:02:45,620	00:02:41,340	00:03:10,692	162086	165620	161340	190692
23	00:02:39,867	00:03:01,734	00:02:55,334	00:02:46,866	159867	181734	175334	166866
24	00:02:40,600	00:03:09,971	00:02:43,790	00:02:52,279	160001	189971	163790	172279
25	00:02:39,973	00:03:09,179	00:03:04,349	00:02:49,849	159973	189179	184349	169849
26	00:02:41,167	00:03:05,323	00:02:57,954	00:03:01,552	161167	185323	177954	181552
27	00:02:42,928	00:03:00,797	00:02:50,274	00:02:47,288	162928	180797	170274	167288
28	00:02:40,448	00:02:53,603	00:02:47,115	00:02:50,592	160448	173603	167115	170592
29	00:02:41,152	00:03:00,646	00:03:05,306	00:02:55,544	161152	180646	185306	175544
30	00:02:42,839	00:03:06,307	00:02:53,717	00:02:42,746	162839	186307	173717	162746